Queen Mary
University of London

Space Decomposition Based Parallelisation Solutions for the Combined Finite-Discrete Element Method in 2D

by

Tomas Lukas

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy and the Diploma of Queen Mary University Of London

School of Engineering and Materials Science

May 2015

I dedicate this thesis to my mum Ruzena, my brothers Josef and Karel and my sisters Ivana and Marcela.

Declaration of originality

The material presented in this thesis is entirely the result of my own independent research under the supervision of Professor Antonio Munjiza. All published or unpublished material used in this thesis has been given full acknowledgement.

Name: Tomas Lukas Date: 8th May 2015

Signature:

List of Publications

Journal Papers:

LUKAS, T.; SCHIAVA D'ALBANO, G. G.; MUNJIZA, A.: Space Decomposition Based Parallelization Solutions for the Combined Finite-Discrete Element Method. Journal of Rock Mechanics and Geotechnical Engineering, Vol. 6, 2014, pp. 607-615

Conference Papers:

LUKAS, T.; MUNJIZA, A.: Parallelization of an Open-Source FEM/DEM Code Y2D. In: 5th International Conference on Discrete Elements Methods (DEM5), London, 2010

MAHABADI, O.K.; LISJAK, A.; GRASSELLI, G.; LUKAS, T.; MUNJIZA, A.: Numerical Modelling of a Triaxial Test of Homogeneous Rocks Using the Combined Finite-Discrete Element Method. EUROCK, Lausanne, 2010

LUKAS, T.; MUNJIZA, A.: Space Decomposition Based Parallelization Solutions for the Combined Finite-Discrete Element Method in 2D. In: Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden, 2013

SCHIAVA D'ALBANO, G. G.; MUNJIZA, A.; LUKAS, T.: Novel MS (MunjizaSchiava) Contact Detection Algorithm for Multi-core Architectures. In: III. International Conference on Particle-Based Methods (Particles 2013), Stuttgart, 2013

SU, F.; TISSERA, S.; LUKAS, T.; MUNJIZA, A.: Use Improved Gradient Descent in Irregular Boundary Conditions in Molecular Dynamics. In: Proceedings of 4th International Conference on Mechanics, Simulation and Control (ICMSC 2014), Moscow, Russia, 2014

Abstract

The Combined Finite-Discrete Element Method (FDEM), originally invented by Munjiza, has become a tool of choice for problems of discontinua, where particles are deformable and can fracture or fragment. The downside of FDEM is that it is CPU intensive and, as a consequence, it is difficult to analyse large scale problems on sequential CPU hardware and parallelisation becomes necessary. In this work a novel approach for parallelisation of the combined finite-discrete element method (FDEM) in 2D aimed at clusters and desktop computers is developed. Dynamic domain decomposition-based parallelisation solvers covering all aspects of FDEM have been developed. These have been implemented into the open source Y2D software package by using a Message-Passing Interface (MPI) and have been tested on a PC cluster. The overall performance and scalability of the parallel code has been studied using numerical examples. The state of the art, the proposed solvers and the test results are described in the thesis in detail.

Acknowledgements

I would like to express my thanks to my supervisor professor Antonio Munjiza for his continuous support and help during my PhD studies. I would like to thank Guillermo Gonzalo Schiava D'Albano for his friendship during difficult times. My thanks go to Mohammad Saadatfar from Australian National University for his friendship and for access to NCI's Raijin supercomputer which made this thesis possible. To my friend Omid Khajeh Mahabadi for his sequential input files. The rest of my acknowledgements go to my friends Fang Su, Richard Tichy, Petr Ferfecki and Jan Brumek, may he rest in peace.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

SCOPE AND LAYOUT OF THESIS

1.1 Scope of Thesis

The Combined Finite-Discrete Element Method (FDEM), originally invented by Munjiza, has become a tool of choice for problems of discontinua, where particles are deformable and can fracture or fragment. The applications of FDEM have spread over a number of disciplines ranging from mining to mineral processing, biomechanics, medical engineering, nanotechnology and energy.

The typical FDEM analysis combines a finite element-based analysis of continua with a discrete element-based transient dynamics, contact detection and contact interaction. Thus, analysis of problems of discontinua, where particles are deformable and can fracture and fragment, becomes possible. This kind of analysis is very expensive in terms of CPU time and enormous computational power is required to simulate systems comprising a large number of deformable bodies. Thus it is difficult to analyse large scale problems on a sequential CPU hardware and parallelisation becomes necessary. Parallelisation efforts for FDEM range from GPU to clusters and desktop multicore hardware.

The main feature of a typical FDEM simulation is a large number of separate bodies (discrete/finite elements) moving and interacting with each other. The distribution of these bodies within the computational domain changes in an unpredictable way during the run of the simulation. If parallelisation is employed, this causes the migration of bodies from one processor (sub-domain) to another and eventually a workload imbalance is created and this in turn decreases the efficiency of the parallel implementation. Thus the dynamic domain decomposition and load balancing (redistribution of bodies among processors) must be employed in order to keep the workload imbalance to a

minimum.

Dynamic domain decomposition based on a geometric approach rather than topological one is especially suitable for parallelisation of FDEM due to the fact that, unlike the Finite Element Method, which assembles global matrices of the system, each element in FDEM is represented by its own local matrix.

A novel space decomposition-based approach for the parallelisation of 2D FDEM aimed at HPC clusters and desktop computers is presented. The developed parallelisation solvers covering all aspects of FDEM are described in detail and the implementation into the open source Y2D software package is presented. The overall performance and scalability of the parallel code has been studied using numerical examples. The state of the art, the proposed solvers and the test results have been described in the thesis in detail.

1.2 Layout of the Thesis

The thesis is organised into several chapters with the following content:

- Chapter 2 provides a short introduction to FDEM and methods of discontinua in general, as well as an introduction to parallel processing. Parallel processing includes an overview of parallel architectures and overviews of main parallelisation languages. Algorithms commonly used to perform domain decomposition and load balancing are also described. Finally some basic concepts of parallel computing including speed-up, efficiency and floating point arithmetic are summarized.

- Chapter 3 provides a detailed description of both the design and implementation of proposed parallel algorithms, including communication, parallel I/O, migration of elements, re-partitioning and load balancing.

- Chapter 4 deals with verification and performance tests of the proposed parallel algorithms. The parallel implementation of the FDEM code is tested on benchmark examples and the performance of the parallel code is studied.

- Chapter 5 contains some application examples of the parallel code. Performance and scalability of the parallel FDEM code is studied on chosen numerical examples (Brazilian disc test, block caving and open pit slope).

- Chapter 6 provides conclusions and discusses possibilities for future work in the field.

Chapter 2

Domain Decomposition Based Parallelisation of Methods of Discontinua

2.1 Introduction

The typical FDEM analysis combines a finite element-based analysis of continua with a discrete element-based transient dynamics, contact detection and contact interaction. Thus, analyses of problems of discontinua, where particles are deformable and can fracture and fragment, become possible. This kind of analysis is very expensive in terms of CPU time and an enormous computational power is required to simulate systems comprising a large number of deformable bodies (millions of particles).

The continuous increase in performance of microprocessors (CPU) by around 50% from 1986 to 2004[16, 142] has led to the development and wide-spread use of methods of Computational Mechanics. It has enabled to solve larger and larger systems at a relatively low cost. The increase in CPU performance per year dropped significantly since 2004.[142] The evolution of Intel processors over the last twenty years is shown in Figure 2.1. For this reason manufacturers of microprocessors started to produce multicore processors in order to achieve continuous increases in performance. Using a multicore processor doesn't automatically mean that the performance of sequential codes will improve. These codes were written for one processor only and the only way to increase their performance lies in parallelisation.[180, 181]

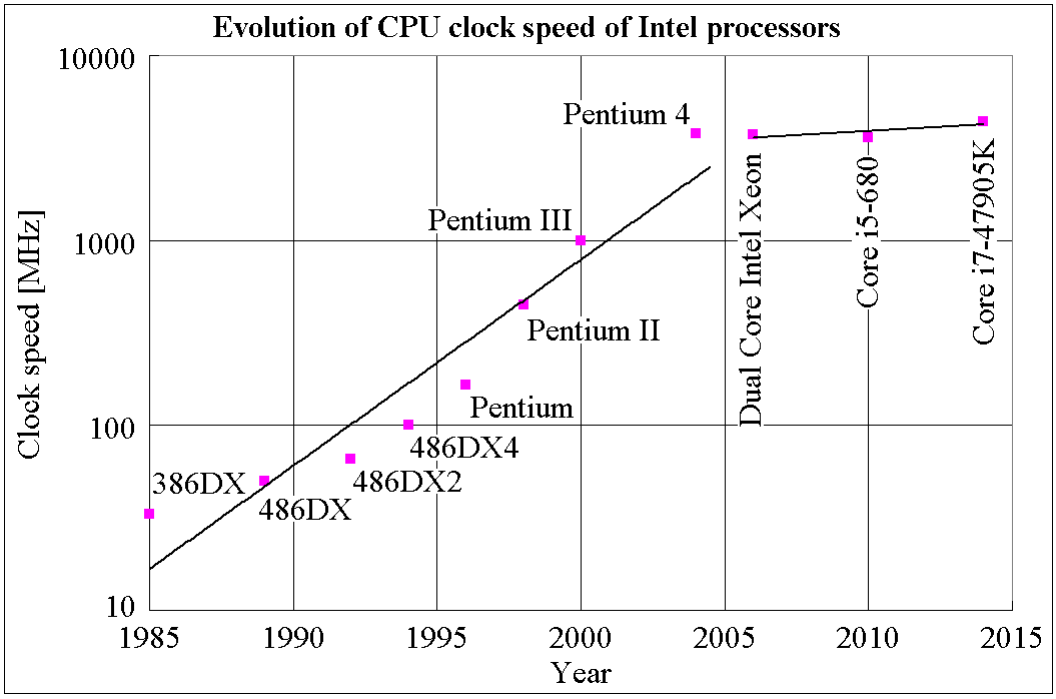


Figure 2.1: Evolution of CPU clock speed of Intel processors. Figure adapted from Munjiza et al.[122] Based on data from Intel[72,71] .

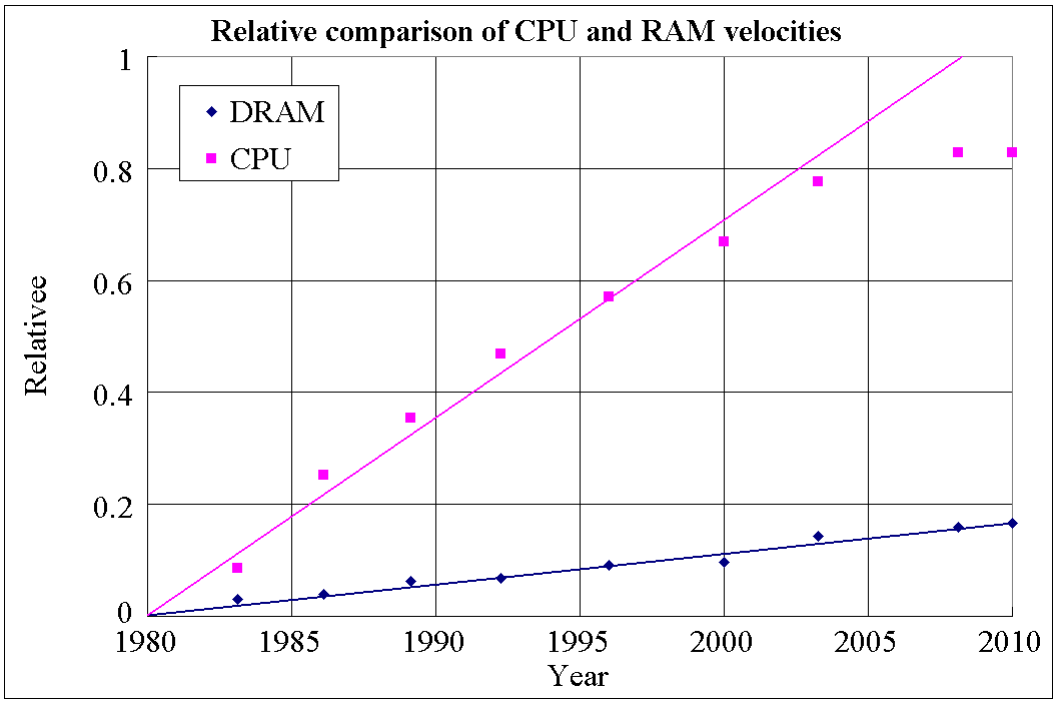


Figure 2.2: Relative comparison of CPU and DRAM velocities. Figure adapted from Borkar et al.[16] .

The increase in speed of Dynamic Random Access Memory (DRAM) has been slower than in the case of CPU, see Figure 2.2. Thus the memory speed became a bottleneck limiting the overall performance of the system. This problem was solved by introducing more than one level of cache.[16]

Together with a slower increase in performance of CPU and DRAM, a continuous decrease of CPU and DRAM prices in recent years can be observed.[122] It stands to reason that parallel architectures, like a High Performance Computing (HPC) cluster, will become even more affordable in the future. Combined with a production of multi-core processors as well as Graphics Processing Units (GPUs), it can be concluded that parallelisation is currently the only reasonable way to increase computational power.

The rest of this chapter provides a short overview of methods of discontinua, parallel architectures and parallel programming languages, as well as an overview of tools and techniques specifically designed for the parallel processing of methods of discontinua.

2.2 Computational Methods of Discontinua

2.2.1 Introduction

Many problems in engineering can be solved by methods based on an assumption that the material can be considered continuous. For example the Finite Element Method (FEM)[212] discretises the continuum by dividing it into small pieces (finite elements) of different shapes and order. Global matrices are assembled for the whole system of finite elements comprising the whole problem The resulting partial differential equations are solved considering the boundary conditions.

A wide range of problems that cannot be solved by continuum methods exist. These include: particle simulations - considering interactions between them, atomistic simulations, presence of joints in a rock, etc. For these problems computational methods of discontinua have been developed.

Methods of discontinua include:

- Discrete Element Method (DEM).
- Molecular Dynamics (MD).
- Smoothed Particle Hydrodynamics (SPH)
- Combined Finite-Discrete Element Method (FDEM)[117, 122]

Discrete Element Method. Everything started in 1971 when Cundall[28] presented the Distinct Element Method in order to solve problems in rock mechanics. Its first industrial application was a simulation of granular assemblies.[29] The basis of the method is the solution of an equation of motion for each particle separately considering forces arising from the interaction between particles. Originally, the particles had a simple geometry and were considered rigid but with development, over time, complex particle shapes and deformability of particles have been added. Further reading on this method can be found for instance in a book written by Jing and Stephansson.[76]

Molecular Dynamics . The basis of the method is very similar to DEM in the sense that the evolution of the simulation is achieved through solving the equation of motion for each particle, in this case atoms and molecules. The interaction between atoms and potential energy are defined by molecular mechanics force fields, for instance Lennard-Jones potential. There is considerable further reading available on this subject.[59,94,164,159,57,13]

Smoothed Particle Hydrodynamics.[99] The method was originally developed by Gingold, Monaghan and Lucy (1977) for solving problems in astrophysics. It is a mesh-free Lagrangian method where the coordinates move with the fluid, and the resolution of the method can easily be adjusted with respect to variables such as density. It has been used, for instance, for simulation of underwater explosions, high-velocity impacts and fragmentation, etc.[185,111,101,152,102,97,43,186]

2.2.2 The Combined Finite-Discrete Element Method

FDEM was developed in 1995 by Munjiza et al.[124] Munjiza wrote a first book on FDEM in 2004[117] and a second book in 2011.[122] The basis of the method is a finite element mesh generated separately for each particle within the simulation. Thus FDEM analysis combines a finite element-based analysis of continua with a discrete element-based transient dynamics, contact detection and contact interaction, so analyses of problems of discontinua, where particles are deformable and can fracture and fragment, become possible. The applications of FDEM have spread over a number of disciplines ranging from mining to mineral processing, biomechanics, medical engineering, nanotechnology, energy or use of green materials in structural engineering.[21,104]

Similarly to DEM, the equation of motion is solved for each element separately.

The governing equation is:

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{C}\dot{\mathbf{u}} + \mathbf{F}_{int} - \mathbf{F}_{ext} - \mathbf{F}_{joint} = \mathbf{0} \tag{2.1}$$

where $\mathbf{M}$ and $\mathbf{C}$ are mass matrix and damping matrix respectively, $\ddot{\mathbf{u}}$ and $\dot{\mathbf{u}}$ are the acceleration and velocity vectors respectively, $\mathbf{F}_{int}$ is a vector of internal forces, $\mathbf{F}_{ext}$ is a vector of external forces (including contact forces) and $\mathbf{F}_{joint}$ is a vector of joint forces. The joint element represents a fracture mechanism called a "discrete crack model" also called a "combined single and smeared crack model" developed by Munjiza et al.[117, 119] in order to introduce a transition from continua to discontinua to FDEM.

The equation of motion is solved by an explicit time integration scheme based on a central difference method. Within each time step the following calculations must be performed:

- evaluation of internal forces based on deformation of particles,
- contact detection, for instance by using Munjiza-NBS contact detection algorithm,[120]
- evaluation of forces arising from contact interaction based on a Penalty function method,[118]
- evaluation of forces from joint elements which act as a bond between finite elements,
- application of external forces and solution of equation of motion for each element separately.

It is beyond the scope of this thesis to provide a comprehensive explanation for each part of FDEM. Munjiza's work provides, not only comprehensive reviews of the problems outlined above, but also of other issues like parallelisation or coupling with Computational Fluid Dynamics (CFD).

2.3 Parallel Architectures

In parallel computing, computer architectures are very often classified by Flynn's taxonomy.[46] It introduces several categories into which all computers can be placed. These are:

- Single Instruction Stream-Single Data Stream (SISD) - a classical von Neumann system[142] can be placed into this category.
- Single Instruction Stream-Multiple Data Stream (SIMD) - these systems execute the same instruction on multiple data items. Vector processors are the best example for this category. Some aspects of SIMD computing are also used in GPUs and thus can be placed in this category, even though they are not pure SIMD systems.
- Multiple Instruction Stream-Single Data Stream (MISD) - SIMD or MIMD systems are more appropriate for parallel computing than MISD systems, thus MISD systems will be omitted in this review.
- Multiple Instruction Stream-Multiple Data Stream (MIMD) - shared-memory systems as well as distributed memory systems belong to this category.

2.3.1 SISD Systems

Von Neumann computer architecture was first proposed in 1945.[187] The central processing unit (CPU) consists of an arithmetic-logic unit (ALU) and a control unit. ALU performs calculations while the control unit controls the execution of the program. Both units in the CPU also have so-called *registers*. Register is a small but very fast memory used for storing information about the execution of the program and currently processed data. CPU is also connected to the main memory which stores both program (instructions) and data. This connection between CPU and memory is called *bus*. The whole system is then connected to the input and output devices, see Figure 2.3.

The important feature of von Neumann architecture is that at one time the CPU can either write to or read from the main memory. It cannot do both at the same time. Thus the performance of the whole system is limited by the speed of the information exchange between CPU and memory. This is referred to as the von Neumann bottleneck.[142]

It is clear from the above that the von Neumann system can execute only one instruction at a time and it can read or write one piece of data at a time. Thus it is a SISD system.

Due to the von Neumann bottleneck, current computers are not manufactured as pure von Neumann machines. Most computers nowadays have another memory called *cache*, which is slower than registers but much faster than main memory and CPU is able to read from and write to cache at the same time.[142]

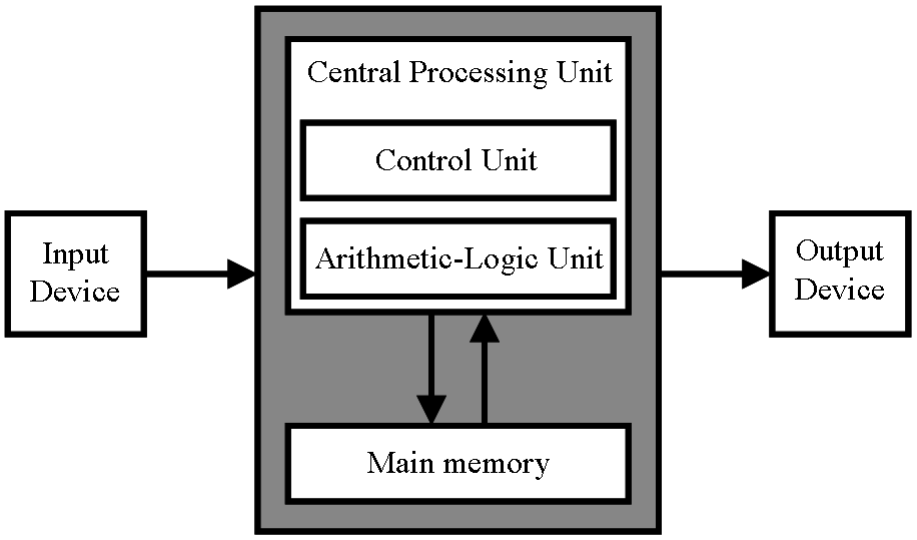


Figure 2.3: Von Neumann architecture. Figure adapted from Pacheco[142] .

2.3.2 SIMD Systems

2.3.2.1 Vector Processors

Vector processors are CPUs capable of performing the same instruction on the whole vector at the same time.[142] This is the main difference between them and conventional CPUs, which perform instructions on one piece of data (scalar). In other words, a conventional CPU must perform a loop over the vector of length n and the vector is processed in n instructions, while vector processor processes the whole vector in one single instruction (no loop is necessary). Thus vector computers are pure SIMD systems. These systems support data parallelism which partitions the data among processors. Processors then perform the same instructions on their assigned data. This type of parallelism is very useful for speeding-up numerical simulations in various scientific disciplines.

Most of the supercomputers in the past (1970s to 1990s) were designed as vector machines. For instance the design of Cray supercomputers was based on vector architecture.[144] Due to the decreasing price and increasing performance of conventional CPUs, vector supercomputers were replaced by PC clusters, comprising large number of conventional CPUs interconnected by network. An overview of vector architectures is presented by Duff.[37]

In the past vector machines were successfully utilised for parallelisation of methods of discontinua, most notably for speeding up Molecular Dynamics (MD) simulations.[8,26,158,155,157,149]

2.3.2.2 GPU

The demand for better, more realistic graphics, particularly for the gaming industry, led to the invention and rapid development of Graphics Processing Units (GPUs). The design of a GPU is quite different from CPU. GPU architecture is SIMD-based in order to achieve high performance through massive parallelism.[130, 134] GPU mostly consists of ALUs, smaller control units and cache while CPU consists of a few ALUs and a larger control unit and cache, see Figure 2.4. CPU design is MIMD-based. Due to this difference, CPU is able to handle a wide variety of tasks in contrast to GPU which is more restricted. Thus, in order to fully utilise the raw processing power of GPU, the program must be carefully designed. This makes implementation of the parallel program for GPU much more complex than in the case of a parallel program written for an HPC cluster. More detailed information on GPUs can be found in literature.[130, 134, 140]

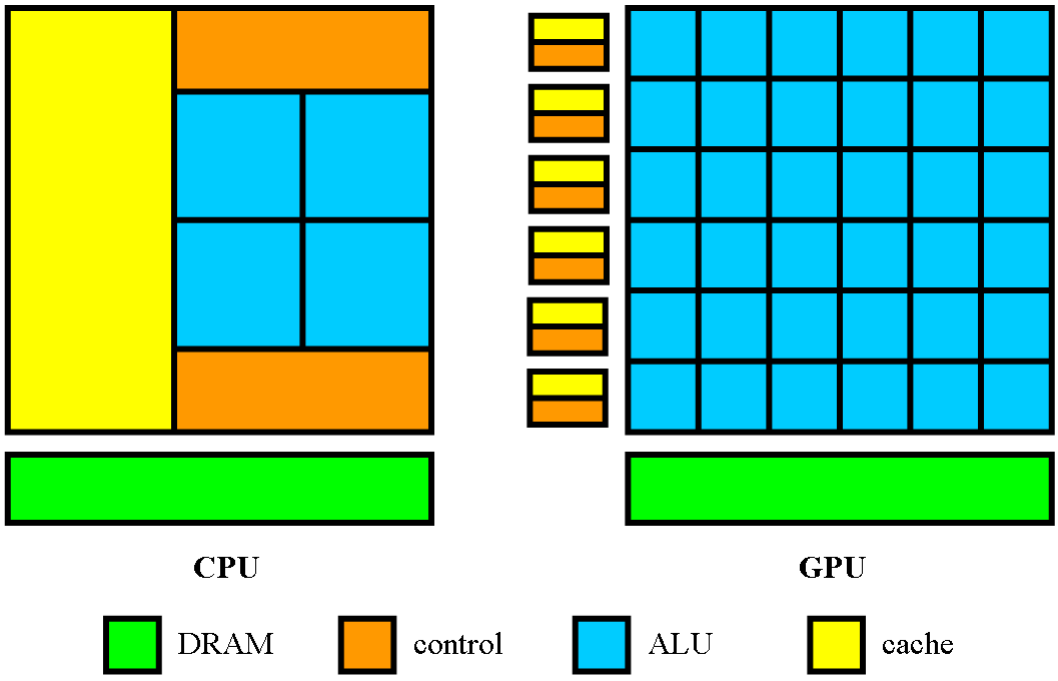


Figure 2.4: Difference between CPU and GPU architecture. Figure adapted from Navarro et al.[130] .

GPUs have been noted in the scientific community because of their high performance and low cost compared with HPC clusters, and they are increasingly being used to speedup numerical simulations, not only in methods of discontinua but in other disciplines as well. GPUs have been successfully used for the parallelisation of MD simulations,[49, 92, 211, 93, 67] Smoothed Particle Hydrodynamics simulations[203, 35, 184] and the Pseudo-Particle Method.[204] An overview of GPU-based MD algorithms is presented

by Stone et al.[178] In Discrete Element Method (DEM), GPUs have been used, for instance, to simulate a powder mixer,[153] powder transportation,[171] digital terrain analysis,[151] tote blender,[161] self-compacting concrete flow simulation,[210] simulation of rotating drum,[208] rigid body transport simulation,[54] mill charge motion simulation[154] and granular media.[55] The GPU-based DEM code, incorporating sliding friction model and thermal conduction model, has been presented by Steuben et al.[176, 175]

Since GPU's available memory for the simulation is limited, there have been attempts to develop a parallel multi-GPU codes. Message-Passing Interface (MPI) is being used for communication between GPUs.[75, 184, 170]

2.3.3 MIMD Systems

2.3.3.1 Shared-Memory Systems

A shared-memory system comprises a number of processors and one main memory which is shared between them. Each processor is connected to the main memory and it can access any location in the memory, see Figure 2.5. Multicore processors are shared-memory systems since each processor contains multiple cores (CPUs) sharing the same memory.[14]

The shared memory systems can be divided into two categories depending on their memory access. *Uniform memory access* (UMA) means, that all processors are able to access the memory in roughly the same time, since all processors are interconnected to the memory through one shared connection. On the other hand, in a *non-uniform memory access* (NUMA) system each processor has its own connection to its block of memory and neighbouring processors are interconnected. Thus, each processor can only access other memory blocks through neighbouring processors. It follows that access to the memory block directly connected to the processor is much faster than access to the remaining blocks. As a consequence, parallel programs for UMA systems are easier to write but, if a parallel program running on NUMA system accesses only directly connected blocks of memory, the memory access is faster and NUMA system can scale to a higher number of processors.

Parallelisation efforts in the field of methods of discontinua directed at the shared-memory systems have been significant due to the widespread use of multicore processors in recent years. These efforts include MD simulations[38, 116] as well as DEM simulations.[133, 12, 200] A parallel implementation of coupled FEM/DEM by using OpenMP, tested by simulating a compression of 3D particle assembly, has been presented by Frenning.[48] FDEM parallelisation aimed at multicore PCs has been presented by

Owen and Feng[138] and by Schiava D'Albano.[168,167] Hopkins and Song[65] carried-out a performance comparison of a parallel DEM code designed for both shared-memory system (using OpenMP) and distributed-memory systems (using MPI). Comparison of MPI-based and OpenMP-based parallelisation of a coupled DEM-CFD code has been done by Amritkar et al.[6]

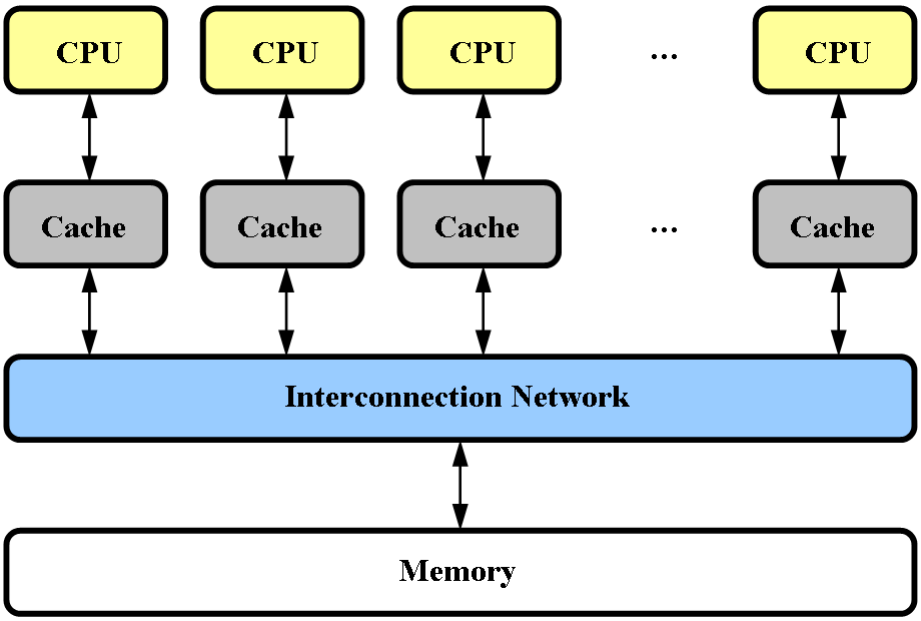


Figure 2.5: A shared-memory system. Figure adapted from Patterson and Hennessy[144] .

Interconnects for shared-memory systems. The interconnection between processors and memory can be either a *bus* or *switched* interconnect. The bus contains multiple parallel communication wires and access control. Bus can be connected to multiple devices which all share access. Multiple cores connected to the memory create a bottleneck, since more than one core can try to access memory at the same time. Then one core must wait until the other one finishes its memory access. For this reason, switched interconnects are increasingly being used, especially for a higher number of cores. An example of switched interconnect is a *crossbar*, see Figure 2.6. If *i*-th CPU tries to access *i*-th memory module, there is no conflict, since the crossbar allows simultaneous connections. Only scenarios resulting in conflict are those where, for instance, CPUs 1 and 2 try to access the same memory module at the same time. The disadvantage of a crossbar is its high price compared with bus.

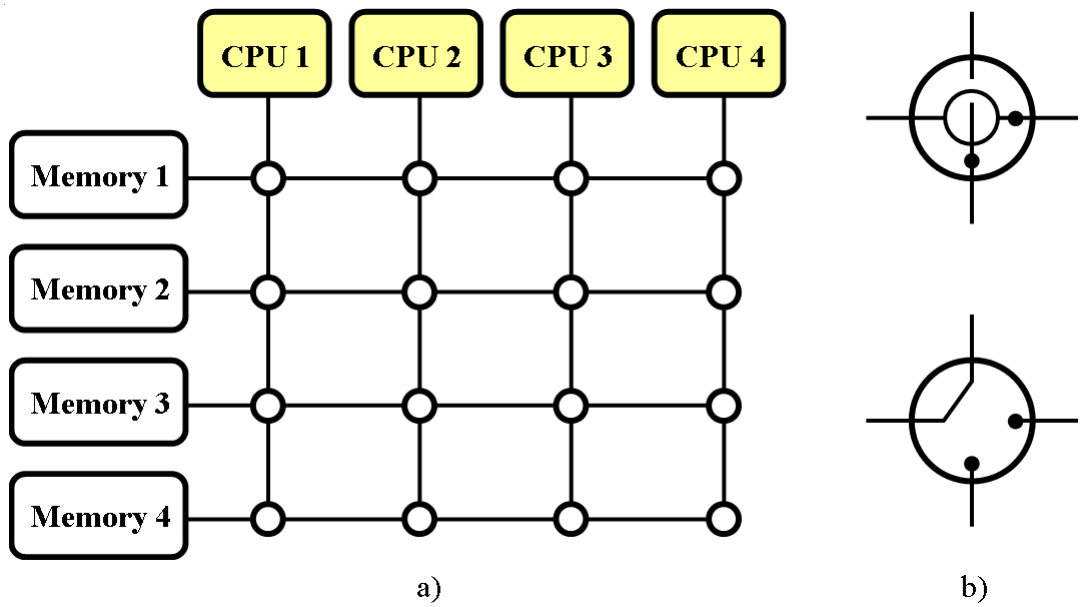


Figure 2.6: a) Four processors connected to four memory modules. Circles in the figure are switches. Lines represent bidirectional links. b) Two possible configurations of internal switches (connected and disconnected). Figure adapted from Pacheco[142] .

2.3.3.2 Distributed-Memory Systems

A distributed-memory system comprises a number of processors, each with its own memory, see Figure 2.7. Processors are interconnected by a network which is used for communication by explicitly sending and receiving messages between processors, so-called *message passing*.

The most widely used example of the distributed-memory system is a HPC cluster. A cluster comprises a number of computers interconnected by, for instance, an Ethernet network. Each computer represents one computational unit, the so-called *node*. Nowadays, nodes in the cluster usually contain multicore processors. Thus, each node is a shared-memory system interconnected by the network into a distributed-memory system. Such a system can be called a *hybrid system*.

The main disadvantage of clusters is the speed of the network connection, which is much slower compared with memory interconnection in shared-memory systems. Therefore, communication overhead is much lower in shared-memory systems, since exchanging data between memory modules is much faster than sending data through the network. A shared-memory system also needs only one operating system and almost all memory is available for the executed program. In clusters, each computer must have its copy of the operating system and executed program.

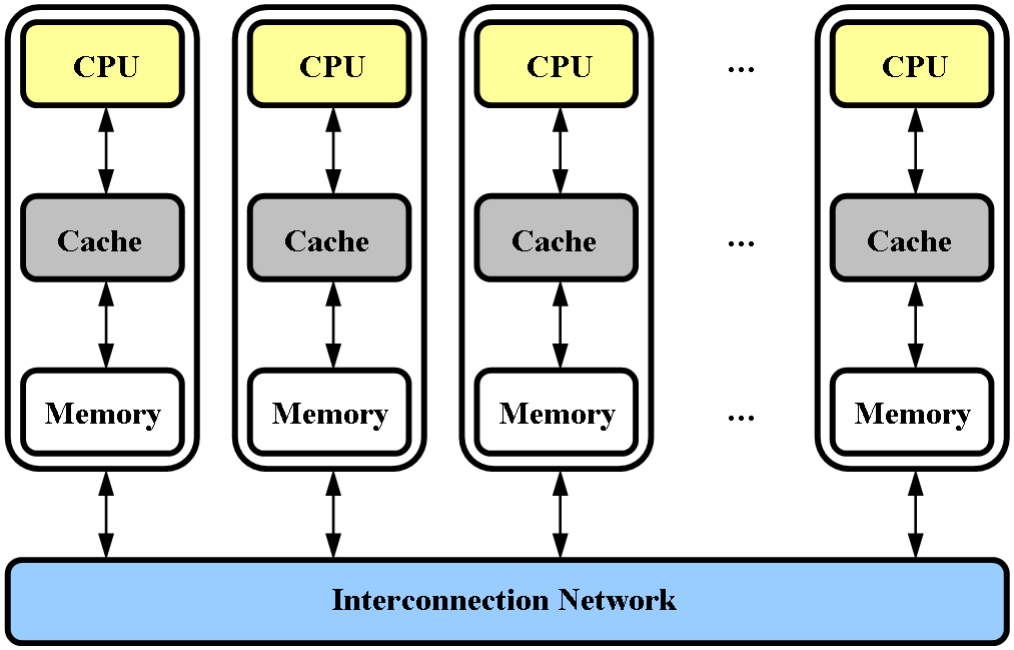


Figure 2.7: A distributed-memory system. Figure adapted from Patterson and Hennessy[144] .

Interconnects for distributed-memory systems. These can be either *direct interconnects* or *indirect interconnects*. In a direct interconnect, switches are directly connected to nodes and to neighbouring switches while, in indirect interconnects, switches are not directly connected to nodes. An example of a direct interconnect can be a *ring*, a *toroidal mesh* or a *hypercube*, see Figures 2.8 and 2.9. The circles in the figures represent switches, each square represents one node (processor(s) and memory) and bidirectional links are represented by lines. It can be observed from the Figures 2.8 and 2.9a, that the ring's switches must handle only two connections (links) while the toroidal mesh must handle five connections. Hence, the switches in a toroidal mesh must be more complex, increasing the price of the network. Both ring and toroidal mesh networks have been used, for instance, for MD parallelisation.[41,40,15] Hypercube (Figure 2.9b) is an example of the practical interconnect which has been used quite often in the past.[173,30,156,63] Generally speaking, hypercubes do not always have to be three-dimensional. In a n-dimensional hypercube, each switch must have n connections plus one connection to the node. Number of nodes p is given by $p = 2^n$.

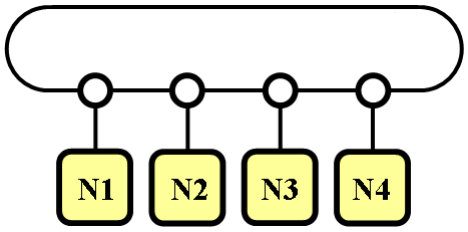


Figure 2.8: A ring. Figure adapted from Pacheco[142] .

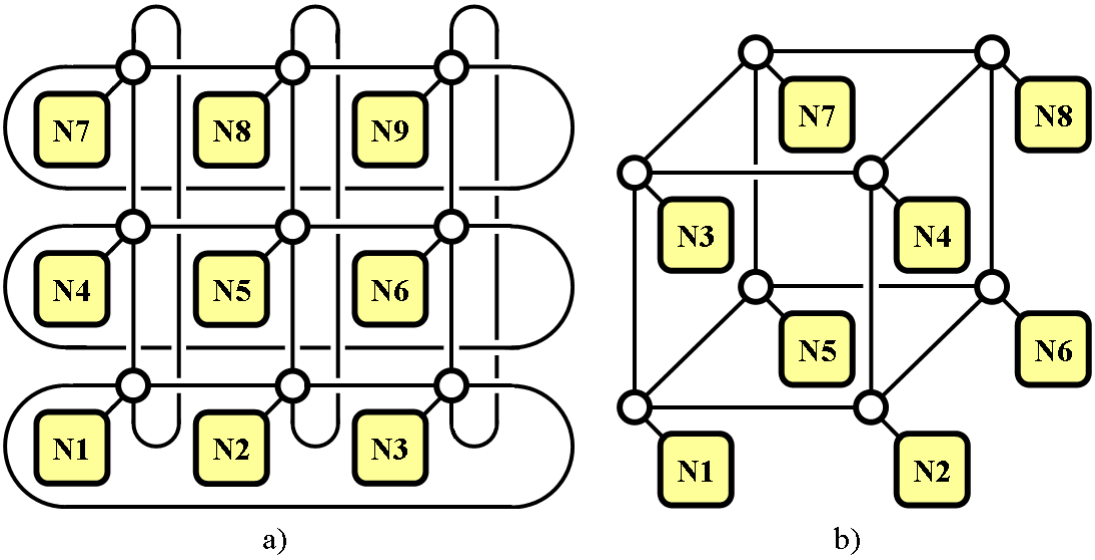


Figure 2.9: a) A toroidal mesh. b) A three-dimensional hypercube. Figure adapted from Pacheco[142] .

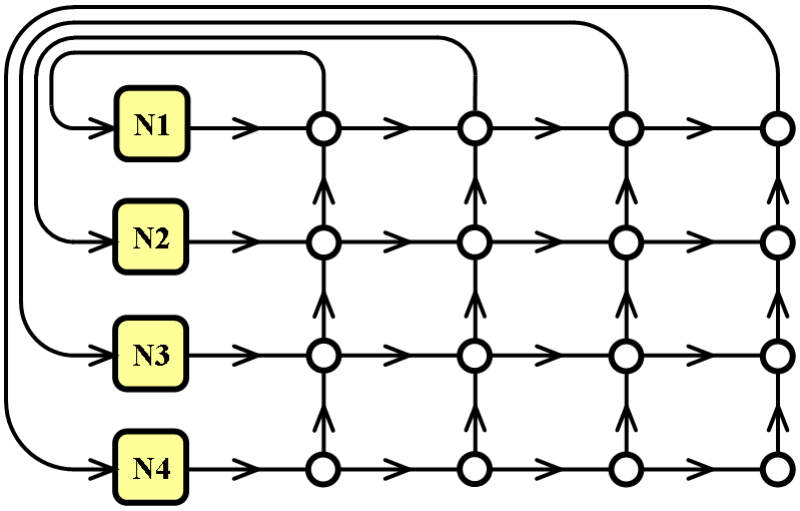


Figure 2.10: A crossbar. Figure adapted from Pacheco[142] .

Figures 2.10 and 2.11 show the *crossbar* and the *omega network*. Both are an example of indirect interconnects. The crossbar for a distributed-memory system is very similar to the crossbar for a shared-memory system. In a distributed-memory system, lines represent unidirectional links which is the main difference between them. Similarly to the shared-memory crossbar, the conflict only arises if two nodes are trying to communicate with the same node at the same time.

Lines in the omega network (Figure 2.11) are also unidirectional links but each switch represents a switch box (two-by-two crossbar), see Figure 2.12. By comparing Figures 2.10 and 2.11, it can be seen that the number of connections and switches in the omega network is smaller than in the crossbar. The crossbar for 8 nodes in Figure 2.11 would comprise 64 switches (p^2, where p is number of nodes). Thus, the crossbar is more expensive than the omega network. The lower price of the omega network comes at a reduced ability to send simultaneous messages. For instance, if node 3 in Figure 2.11 sends a message to node 7 then node 4 cannot send a message to node 8 at the same time.

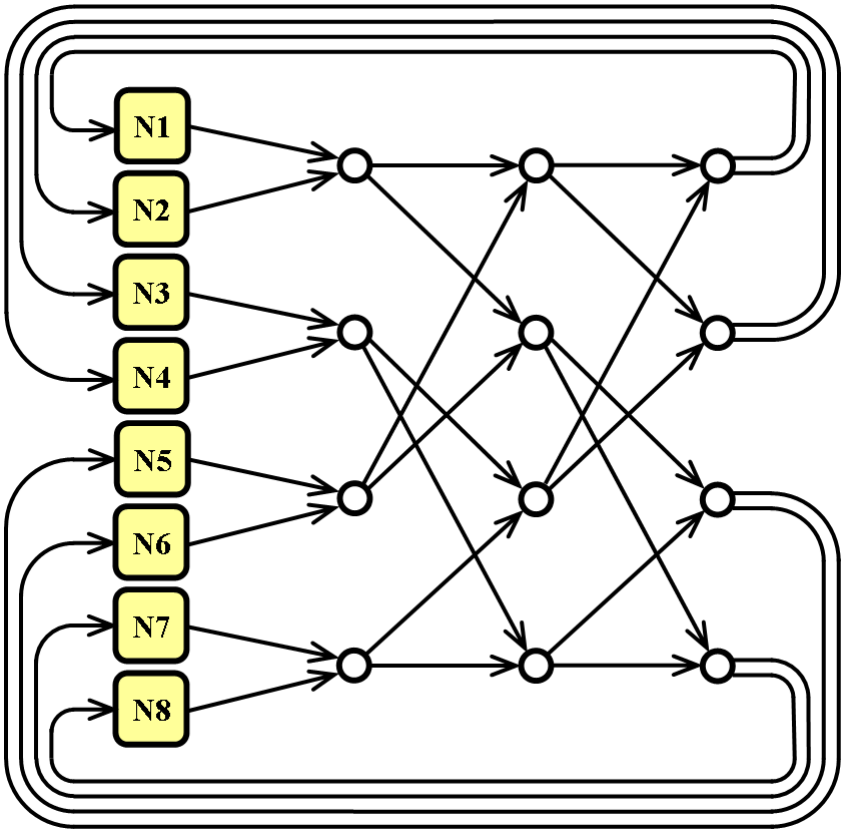


Figure 2.11: An omega network. Figure adapted from Pacheco[142] .

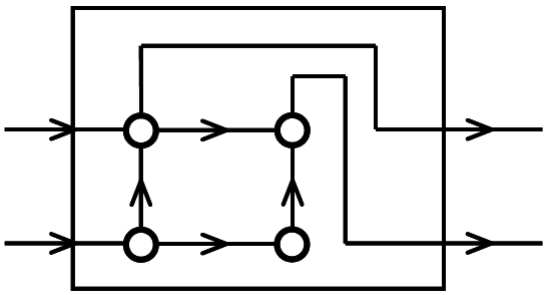


Figure 2.12: A switch box in an omega network. Figure adapted from Pacheco[142] .

The performance of any network is usually described by two values. The first value is a *bandwidth* of a link which is the speed of data transfer in a link (in Mb/s). The time needed to transfer data between two nodes in a distributed-memory system can be calculated from the bandwidth of a link as follows[142]

$$T = l + n/b \tag{2.2}$$

where T is the time needed to transfer n bytes of data, l is the network latency and b is the bandwidth of the link. The network latency is a difference between the time when the first node starts to send data and the time when second node starts to receive the data.

The equation 2.2 can be used to calculate the transfer time of any interconnect (in a shared-memory or distributed-memory system) regardless of which devices it is connecting to (main memory and hard drive, memory and cache, cache and register) since any interconnect has always some latency and a bandwidth.

The second value describing the network performance is a *bisection bandwidth* which can be calculated by multiplying bisection width by bandwidth of links used in the network. To calculate the bisection width for any network, the system is divided by a line/plane in two halves, each containing half of the system's nodes. Then the sum of all the links which were crossed by the line/plane gives the bisection width. In the case of an asymmetric network topology, the system is divided in such a way, that the worst possible bisection width for the network is obtained. The bisection bandwidth for network types described above is summarised in Table 2.1.

Two important points should be noted while choosing the target MIMD system at which the parallelisation will be aimed. The complexity of programming for the shared-memory systems, compared with distributed-memory systems, is reduced since all the data is saved in the shared memory. The main limitation for these systems is the available number of processors and amount of RAM memory. As mentioned above,

with an increasing number of cores connected to the bus, the probability of conflict over access to the bus increases as well. A crossbar connecting a high number of cores to the same number of memory modules would be very expensive. On the other hand, HPC clusters comprising thousands of nodes are readily available. As a consequence, the distributed-memory systems are the logical choice for computationally intensive simulations with high memory requirements.

Network	Number of switches [-]	Bisection width [-]
Ring	p	2
2D Toroidal mesh	p	$2\sqrt{p}$
n-dimensional hypercube	$p = 2^n$	$p/2$
Crossbar	p^2	p
Omega network	$2p\log_2(p)$	$p/2$

Table 2.1: Number of switches and bisection width for different network types, where p is number of nodes (processor-memory pair). Data taken from Pacheco[142] .

Practically any simulation of any method of discontinua requires vast computational power. Thus large problems can be run only on distributed-memory systems by utilising a domain decomposition approach. For this reason the main parallelisation efforts have been directed at these systems. Parallel MD codes utilising replicated data strategy for atom (task) decomposition[136,146,148,182] or force decomposition[19,24,96,127,149] as well as spatial domain decomposition[23,31,74,73,81,90,115,128,137,4,131] have been developed. Parallel DEM codes as well as parallel FDEM codes designed for distributed-memory systems are predominantly using a geometric domain decomposition approach[36,53,191,192] even though a few examples using graph partitioners do exist.[79,138] Parallelisation of DEM has been used to speedup simulations of liquid-particle flows,[7] industrial granular flows,[25,166] gas-solid flows,[103] investigation of particle mixing behaviour,[114] simulation of poly-dispersed granular material[78] and coupled DEM/CFD simulation of a gas-fluidized bed.[183]

As mentioned above clusters nowadays are usually hybrid systems built from shared-memory nodes connected by a network. Some attempts have been made to design a parallel DEM code[64] and a coupled DEM/CFD codes[100] by a hybrid MPI-OpenMP approach.

2.4 Short Overview of Parallel Programming Languages

2.4.1 Introduction

The development of parallel hardware and its widespread use for scientific computation was accompanied by the development of parallel software tools. This is due to the fact, that running a sequential code on, for instance, a multicore machine will not lead to the performance improvement. In order to utilise the parallel architecture of the machine, the parallel version of a sequential program must be written using some available parallel programming language. The choice of the parallel programming language depends mainly on the hardware of the parallel machine.

The main parallel programming languages include:

- Compute Unified Device Architecture (CUDA)[89, 70, 132] is a programming language for writing parallel programs for GPUs. It was developed by NVIDIA[2] and the parallel program developed by using CUDA can be run only on NVIDIA GPUs. It provides an extension to programming languages like C, C++, Fortran, etc. It provides a developer with the means to define so-called *kernel* functions. Unlike standard C functions which are run only once, each kernel function is run N times concurrently on N CUDA threads. Thus, the main difference from the CPU is the fact that GPUs support running thousands of threads in parallel. CUDA is a low-level programming model since the developer using CUDA has full control over threads, all the memory (global, shared) and synchronization.

- Open Computing Language (OpenCL)[125, 134, 56, 34] is a programming framework supporting parallel programming for heterogeneous systems comprising CPUs, GPUs and/or other processors, e.g. field-programmable gate arrays (FPGAs). The framework comprises "a language, application programming interface (API), libraries and a runtime system to support software development".[126] It supports both task-based and data-based parallelism. OpenCL is developed and maintained by the Khronos Group. OpenCL can be classified as a low-level programming model[9] since the programmer must take care of issues like coordination of computation, detailed access to the memory, etc.

- OpenMP[20, 18] is an application programming interface (API) enabling the developer to write parallel programs for shared-memory systems. It was introduced in 1997 by the *OpenMP Architecture Review Board* (ARB). The first version provided support only for Fortran but since that time support for C and C++ has been

added. It is not a programming language since its directives are simply added to the sequential C, C++ or Fortran program. These directives specify division of work among threads and access to the shared memory, while the details of the parallel program are solved by a compiler automatically. Thus OpenMP can be classified as a mid-level programming model.[9] OpenMP's popularity can be attributed mainly to the widespread use of multicore processors and its relative simplicity of programming (at least at the beginner's level).

- Parallel Virtual Machine (PVM):[51] As the name suggests, PVM takes a heterogeneous system of computers (desktops, workstations, clusters) and uses them to create a single virtual parallel machine. Computers in the system can be running Unix and/or Windows operating systems and the whole system must be interconnected by a network. The program execution is coordinated through sending and receiving messages. Thus, parallel programs developed in PVM are using a message-passing programming model. PVM supports C, C++ and Fortran. PVM is a software package which contains a so-called daemon (pvmd3) which enables the creation of the virtual machine and a library of routines for creating a parallel program. The first version of PVM, which was never publically released, was created at Oak Ridge National Laboratory in 1989 as a part of Heterogeneous Distributed Computing research project. The next version (PVM 2.0) was written at the University of Tennessee and it was released to the public in 1991. The latest version is PVM 3.4.6, released in 2009. PVM was widely used in the first half of 1990s but since then its popularity has been steadily declining due to the release of much more successful MPI.

- Message-Passing Interface (MPI)[58, 141, 1, 82] became a de facto standard for message-passing parallel programming model since its original release in 1994 (MPI-1). MPI-2 was released in 1997 and the latest version is now MPI-3 which was released in 2012. A short introduction to MPI is provided in the next chapter. MPI is developed and maintained by the MPI Forum.

Some other parallel programming tools exist, but as none of them reached the widespread use of the ones listed above, they are out of the scope of this chapter.

2.4.2 Message-Passing Interface

MPI is a message-passing API aimed originally at distributed-memory systems but, with the increasing number of multicore computers being used, the support for these

was also added. It provides a library which functions as an extension to C, C++ and Fortran programming languages allowing the programmer to define the communication and coordination between processes by explicitly sending and receiving messages (MPI-1). The MPI-2 added support for such issues as parallel input/output (I/O) operations, dynamic creation and management of processes and remote memory operations. The MPI-3[47] is addressing issues like non-blocking collective communication, one-sided communication and the support for Fortran 2008.

At the beginning of the execution of the developed parallel program, the specified number of *processes* is started. Each process is assigned a unique *rank* and its piece of data of the simulated problem. Processes then coordinate their work by sending and receiving messages. Usually the number of processes is equal to the number of processors, but this does not always have to be true. It is possible to run a couple of processes on a single processor. It follows that:

$$N_{process} \geqq N_{processor} \tag{2.3}$$

where $N_{process}$ is the number of processes and $N_{processor}$ is the number of processors. In the rest of this thesis it is assumed that each processor is assigned only one process, so both words are used in the text interchangeably.

If the parallel code is executed on a shared-memory machine, then MPI creates a private address space located in the shared memory for each process and the message-passing is simulated by sending data between these address spaces.[142]

It is out of the scope of this thesis to provide a complete description of all MPI functions. A complete and in-depth description can be found in numerous MPI books and online tutorials.[58, 141, 1, 82, 87] The following subsections of this chapter provide a quick introduction to some key MPI concepts which are needed for a better understanding of proposed FDEM parallelisation solutions described in Chapter 3.

2.4.2.1 Communication

As mentioned above, message-passing means explicitly sending and receiving messages. It is up to the programmer to define what type of data and how much data to send and receive. In general two main types of communication exist: point-to-point and collective.

Point-to-point communication. This type of communication takes place between two different processors *i* and *j*. If processor *i* sends a message, processor *j* must receive the message. The basic functions to accomplish both are *MPI_Send(parameters)* and

MPI_Recv(parameters). Parameters in both functions are very similar. The user has to specify what data to send (buffer, size of the message and type of data - integer, string of characters, derived datatype, etc.) and each message must be uniquely identified (tag, source/destination and communicator).

A *buffer* on a sending processor is a pointer to the block of memory (array, number, etc.) which will be sent, while on a receiving processor it is a pointer to the empty memory space of matching type where the received message will be saved. The *size* of the message on a sending processor must be exactly equal to the amount of data being sent. Receiving processor usually has no information about the size of the message, unless the message contains only one object. In case of an array/string of characters, the receiving processor should pass to the function a maximum size of the array/string.

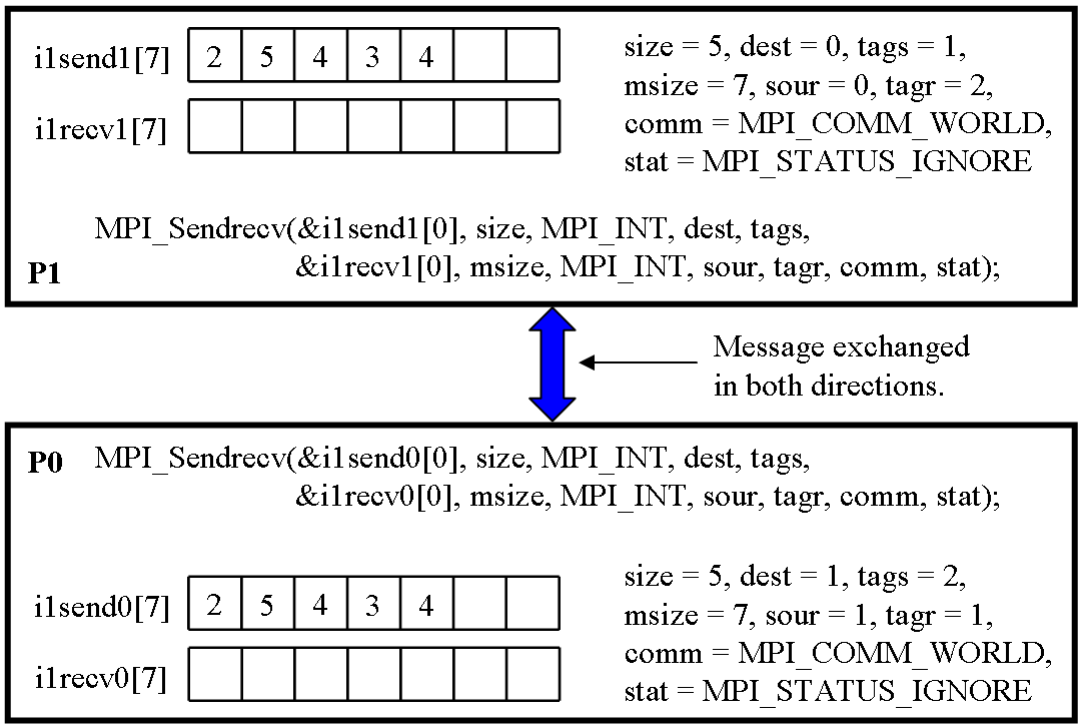


Figure 2.13: MPI_Sendrecv() operation between processors 0 and 1. Arrays i1send0 and i1send1 are received and saved into i1recv1 on processor 1 and into i1recv0 on processor 0, respectively.

The sending and receiving processors must specify *destination* and *source*, respectively. These are the ranks of receiving and sending processors. Each message should have a unique ID in case of multiple messages between the same processors. This ID is called a *tag*. The last parameter is a *communicator*. A communicator in MPI means a group of processors which can send and receive messages between each other. A com-

municator can be user defined or a predefined MPI_COMM_WORLD communicator can be used. The group in MPI_COMM_WORLD includes all processors on which the program is executed. Function MPI_Recv() has one additional parameter: *status*. Status is a structure holding some key information about the message: MPI_SOURCE, MPI_TAG and MPI_ERROR.

As mentioned above, the time needed for sending any message includes a network latency. When two processors need to send/receive one message to/from the other, MPI provides a function *MPI_Sendrecv(parameters)*. By using this function the latency is halved because the connection between the two processors must be established only once. The parameters for this function are a combination of parameters for functions MPI_Send() and MPI_Recv(). Since both the sending and receiving buffer must be specified, the size of sending and receiving message must be specified, etc. The use of this function is illustrated by sending an array of 5 integers from both processors, see Figure 2.13.

Collective communication. Unlike point-to-point communication all processors included in the communicator participate in the communication. The message is exchanged globally between all processors in case MPI_COMM_WORLD is used.

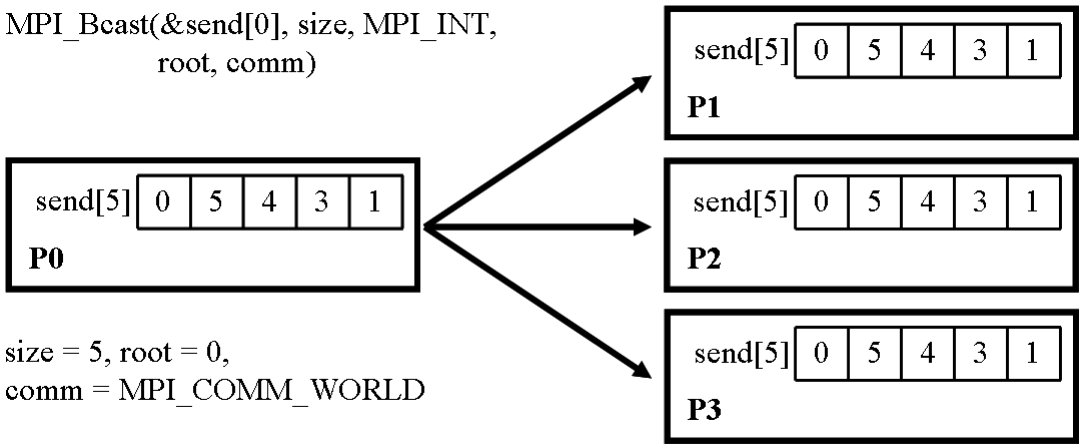


Figure 2.14: Processor 0 distributes an array of five integers to remaining processors by using MPI_Bcast() function.

Broadcast function *MPI_Bcast(parameters)* is a typical example of a collective communication. This function sends data from one processor, called *root*, to all remaining processors in the communicator. Since one message is exchanged between all processors the parameters passed to the function must be the same on all processors. Similarly to point-to-point communication, the buffer, the size of the message and the type of data must be specified. The remaining two parameters are the rank of the root

processor and a communicator. Figure 2.14 illustrates the use of broadcast function by distributing an array of five integers from processor 0 to the remaining processors.

MPI_Allgather(parameters) is an MPI collective communication function which takes specified data from each processor in the communicator, combines them and saves the result to the receiving buffer on all processors. The user has to specify the size of the message and the type of data for both sending buffer and receiving buffer. The use of MPI_Allgather is illustrated by gathering an array of 5 integers saved on 4 processors, see Figure 2.15.

All communication functions described above belong to the group of so-called *blocking* functions. If, for instance, processor *j* calls MPI_Recv() but processor *i* still did not call MPI_Send(), then processor *j* will wait until the message from processor *i* is received. Thus, the use of blocking communication provides the user with means to synchronize the execution of the program on different processors. As a consequence, the use of blocking communication prohibits overlapping communication and computation. This can be achieved by using a *nonblocking* communication.

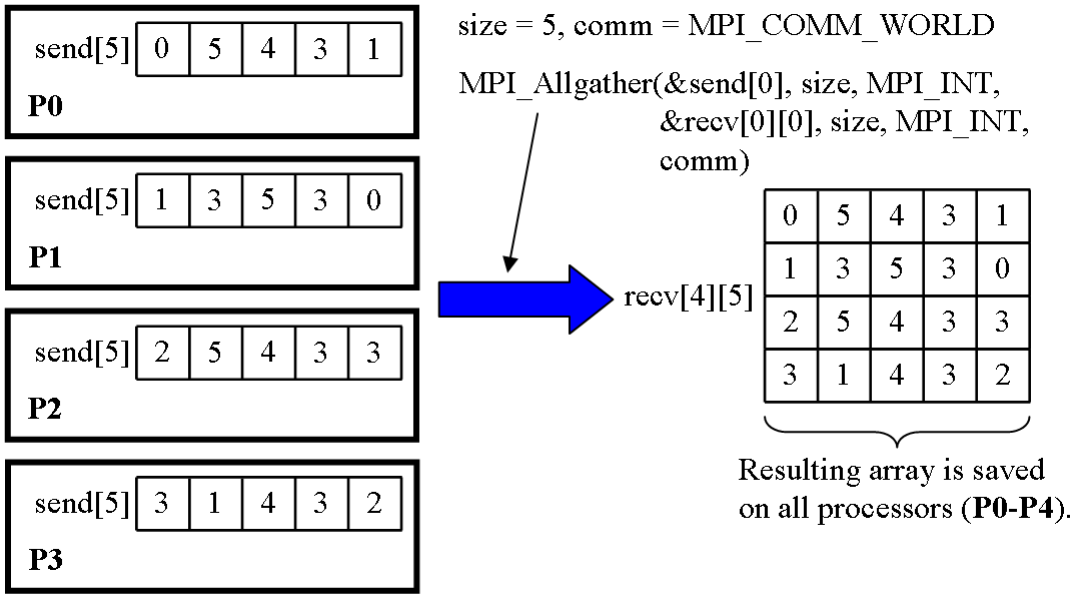


Figure 2.15: MPI_Allgather() operation between 4 processors. An array of five integers is gathered from each processor and the resulting array is saved on all four processors.

Derived datatypes. In all the examples from the Figures above the message contained an array of data of one type which was saved in contiguous memory locations. It is very often necessary to send non-contiguous data items or data items each with a different type. For these cases MPI provides functions to build a so-called *derived*

datatype which in essence keeps an address of each data item and its type. Derived datatype can be then passed to any communication function as a type parameter. The derived datatype for n data items can be expressed as

$$(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1}) \tag{2.4}$$

where t_i is the type of data of i-th data item and d_i is the displacement from the first data item in bytes. When the derived datatype is passed to the communication function, the starting address of the first data item is provided. The i-th address of the i-th data item can be calculated by using the displacement d_i.

MPI_Type_create_struct(parameters) is the most general function for building a derived datatype. The user must pass the following parameters to the function: *count* (how many data items will be sent), *block_lengths[]* (each data item does not have to be 1 number, it can be an array of numbers of the same type), *displacements[]* (displacement of each data item from the first item), *typelist[]* (data type of each data item) and *new_mpi_t* (a pointer to the newly created datatype).

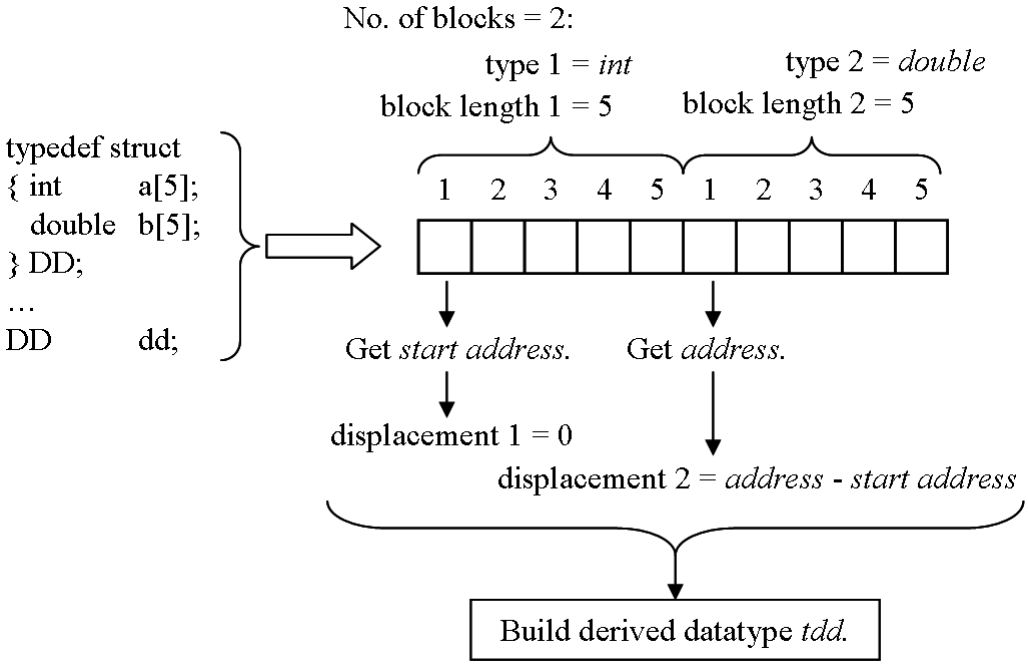


Figure 2.16: Parameters needed to build a derived datatype *tdd* from a structure dd which contains an array of 5 integers and an array of 5 numbers of type double.

In order to find a displacement of each data item, the address of each must be found first. MPI provides a function *MPI_Get_address(data_item, &address)* which returns an address of an argument *data_item* and saves it into an argument *address*. When

all addresses of all data items are found, displacement from the first data item can be calculated for each item (displacement of the first one is equal to zero). After the new datatype is built, it cannot be used until a MPI function *MPI_Type_commit(new_mpi_t)* is called.

Figure 2.16 illustrates the building of a derived datatype *tdd* for a user-defined structure dd. The structure contains two arrays, 5 numbers long, of types int and double. Since the C guarantees that both arrays in the structure will be stored in contiguous memory locations, it is possible to view them as one long array and to calculate the displacement of the second array from the beginning of the first one. All the parameters needed for building the derived datatype *tdd* are given in the Figure 2.16.

Since it is fairly expensive to build the derived datatype, it should ideally be re-used many times during the execution of the program. All the main communication events in the proposed FDEM parallelisation strategies use derived datatypes, see Chapters 3.10.2 and 3.12.

2.4.2.2 Parallel Output Operations

MPI provides support to perform parallel Input/Output (I/O) operations from version 2. It should be noted that MPI provides support only for unformatted binary files. Only the basic MPI output functions will be explained since only the output is parallelised in the proposed FDEM parallelisation.

MPI_File_open(parameters) function opens the file with a specified name. The first parameter in the function is a communicator. This means, if the communicator is MPI_COMM_WORLD, all processors have access to the opened file. The next arguments are a file name, a mode of access (read, write or both) and *info* argument (can be omitted by using MPI_INFO_NULL). The last one is a *file handle* returned by the function. Function *MPI_File_close(file_handle)* closes the specified file.

A function MPI_File_write(parameters) writes specified data which are saved in a buffer into output file. The parameters for the function are: *file handle*, *buffer* containing output data, *count* of data items, *type* of data and a *status* which can be omitted by using MPI_STATUS_IGNORE).

Since the output file is opened on all processors specified by the communicator, each processor can write into the file. Thus, the file is shared among processors and the output file consists of contributions from all processors in the communicator. It is up to the user to set a so-called *file view* for each processor, which is a part of the file accessible only by a specified processor. The function *MPI_File_set_view(parameters)*

allows the user to set the file view for each processor.

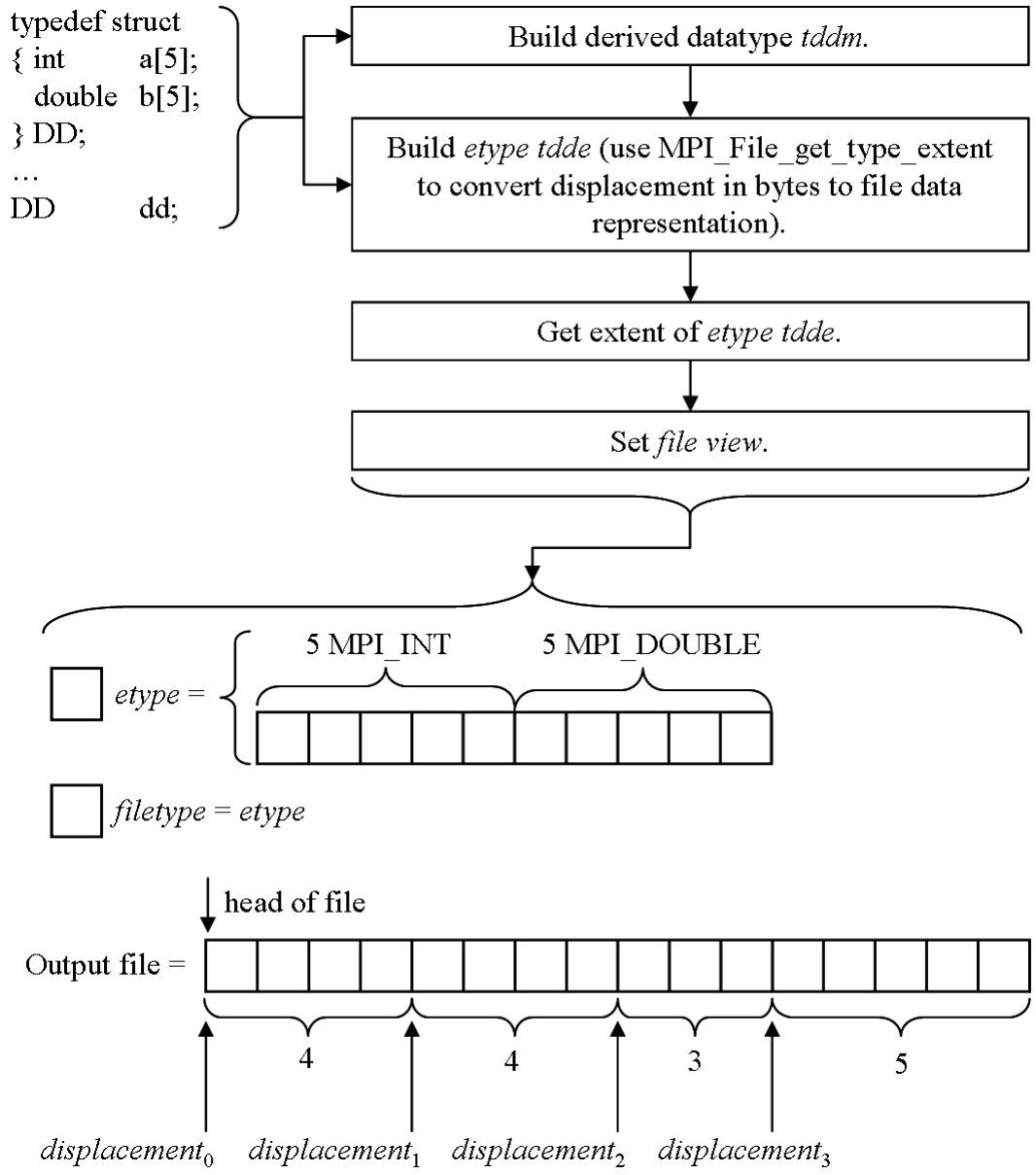


Figure 2.17: Each processor writes into the output file an array of a structure dd which contains an array of 5 integers and an array of 5 numbers of type double. Derived datatypes *tddm* and *tdde* are built for the structure and an extent of *etype tdde* is found. *File view* is set for each processor by using numbers of data items for each processor and the extent of etype. Numbers of data items of *etype tdde* on processors 0, 1, 2 and 3 are 4, 4, 3 and 5, respectively.

The file view comprises three values: *displacement*, *etype* and *filetype*. Displacement is a number of bytes from the beginning of the file. Etype is the unit of data access and filetype points to the part of file visible to the processor as well as holding the in-

formation about the type of data. Both etype and filetype can be derived datatypes. The arguments of the function are the *file handle*, *displacement*, *etype*, *filetype*, *data representation* (for instance to specify that the data will be written in the file as they appear in the memory) and *info*.

The main advantage of constructing a file view from a derived datatype is the possibility to write into file a set of noncontiguous data. The use of the derived datatype introduces a problem, since the displacement in the datatype is in bytes (memory representation) but, for writing into a file, it must be expressed in the file data representation. This can be done by using a function *MPI_File_get_type_extent(parameters)*. The parameters of the function are a *file handle* and a *derived datatype*. The last parameter is an *extent* which is returned by the function.

To summarise, by using the above functions, it is possible for every processor to write its output data in parallel into one shared output file. The use of the functions above is illustrated by writing an array of a structure into the output file. The structure contains an array of 5 integers and an array of 5 numbers of type double, see Figure 2.17.

2.5 Parallel Processing of Methods of Discontinua

2.5.1 Introduction

A typical approach to parallelisation in the field of methods of discontinua is to divide the computational domain into many smaller pieces (sub-domains). This approach is referred to as domain decomposition. The two main objectives of parallelisation are to have a balanced workload (load/CPU time on each processor ideally being the same) and that the amount of data exchanged between processors during communication should be as small as possible.

The main feature of any method of discontinua is a large number of separate bodies (discrete elements, particles, atoms etc.) moving and interacting with each other. The distribution of these bodies within the computational domain is changing in an unpredictable way during the run of the simulation. If the parallelisation is employed, this causes the migration of bodies from one processor (sub-domain) to another and, eventually a workload imbalance is created. This, in turn, decreases the efficiency of the parallel implementation. Thus, the dynamic domain decomposition and load balancing (redistribution of bodies among processors) must be employed in order to keep the workload imbalance to a minimum.

In the rest of this chapter an overview of some of the most commonly used dynamic domain decomposition techniques is presented and their main features are discussed.

2.5.2 Dynamic Domain Decomposition and Load Balancing

Two main approaches to dynamic domain decomposition exist: geometric and topological.[61] The partitioning of the computational domain in geometric methods is based on dividing the domain according to the location of all bodies within the domain while the partitioning in topological methods is based on exploiting the connectivity of interactions. From this connectivity, a graph can be constructed, expressing the distribution of load within the domain.

Independently from geometric or topological methods stands a master/slave approach. The main idea of this parallel programming model is to keep one processor (master) distributing the work between the remaining processors (slaves). This approach is quite easy to implement. It is suitable for problems where tasks assigned to slaves can be solved independently from other processors and the computation on each slave can be performed asynchronously from other processors. This property makes the master/slave approach unsuitable for methods of discontinua.

2.5.2.1 Geometric Methods

In a typical problem of any method of discontinua the bodies located within a computational domain interact with each other only if they are located in a close proximity. Thus, if a domain is sliced by a line in 2D or by a plane in 3D into two sub-domains, each sub-domain has bodies located within its boundaries. In addition, both sub-domains also have to keep copies of bodies intersected by the slice and, during the run of the simulation, the communication between sub-domains is needed for these copies.

Recursive Coordinate Bisection (RCB) method. This method uses orthogonal planes to coordinate axes in order to partition the domain recursively. The domain is first partitioned along the x direction in such a way that the resulting two partitions will each have the same amount of computations assigned. Then these two partitions are divided along the y direction independently from each other following the same rule. The resulting four partitions are again divided along the x direction, etc. until the specified number of partitions is reached. This method was first introduced by Berger and Bokhari[11] as a static load balancing algorithm but it can also be used for dynamic load balancing. This algorithm is implicitly incremental since the small change in the

position of the object within the domain will introduce only a small change in partitioning. Therefore, the following redistribution of elements between sub-domains comes at a smaller cost, since only a small amount of elements located in close proximity of borders needs to be sent to another processor. In addition, the RCB algorithm is easy to implement. For all these reasons it is often used for load balancing. It has been successfully applied, for instance, to Pseudo-Particle Modelling (PPM),[105,194] SPH[147] and FDEM simulations.[191,192]

Unbalanced Recursive Bisection (URB) method. This method is a modification of the RCB algorithm presented by Jones and Plassmann.[77] The original RCB algorithm creates partitions by dividing the load in half, with no regard to the shape of the resulting domain. If a rectangle is much longer along one direction than the other, the communication cost along that direction will be increased. Thus, the URB algorithm considers the geometric shape during the partitioning to avoid rectangles with large aspect ratios. Since this method is a modification of the RCB algorithm, the URB algorithm is incremental as well.

Recursive Inertial Bisection (RIB) method. This method was first introduced by Williams.[201] The domain is divided by planes, which are orthogonal not to a coordinate axis but to a principal axis of inertia calculated from the combined mass of all bodies comprising the simulation. All bodies are assumed to be point masses. This approach gives partitions of better quality than RCB and URB but the method is not incremental and it is a bit more expensive.[61]

Other purely geometric load balancing algorithms exist but never became as popular as the methods described above. These include partitioning of the domain by circles/spheres[112] or partitioning into Voronoi cells.[42,177]

Octree Partitioning.[27,45,113] This method is also known as Space-Filling Curve (SFC).[145] To construct a so-called *octree*, the domain is divided into 8 regions (in 3D) by splitting each axis in half (regardless of positions of objects within domain). Then the same is done for each region if it contains multiple objects until each region contains one object. The information about relationships between these regions is stored in the octree. The root of the octree is the original global domain. When the global domain is divided into 8 regions, 8 child octants of the root are created, etc. By performing a tree search, the global ordering of an object can be found and this information can be used for partitioning. Thus, each resulting partition is a combination of many small regions and the partition shape is not a simple box. SFC partitioning is an incremental algorithm. This method was successfully applied to, for instance, adaptive mesh refinement.[135,143]

2.5.2.2 Topological Methods

Local methods. As the name suggests, the workload is balanced only on a current processor and its neighbouring processors. Each processor is assigned its own neighbourhood. These methods are iterative and, in one iteration, load is balanced within each neighbourhood. The whole global grid can be balanced during a number of iterations since assigned neighbourhoods are overlapping by design.

Each iteration is CPU inexpensive since it is performed only on a small group of processors and, if the method is used for local improvement around the processor with highest workload, then the performance of the whole system can be improved at a small cost. On the other hand, to achieve an even workload over the whole global grid of processors, many iterations may be needed. Also, using a local method leads to lower partition quality.

Local methods are incremental and they scale well with increasing number of processors. Local mesh refinement during the run of the simulation is one of the cases for which these methods are well suited. For a typical simulation in methods of discontinua, involving a majority of objects moving and migrating between processors, it may be better to use a global method.

It should be noted that unlike global methods, local methods can be asynchronous. If the method is synchronous, then the behaviour of the system is similar to the global method where less loaded processors must wait for processors with a higher workload. The load balancing is performed when the system is synchronized.[27,32,189,190,196] If the method is asynchronous, less loaded processors can start load balancing within its neighbourhood while processors with a higher workload are still computing. Asynchronous methods are much more difficult to implement. They can be implemented by using threads[22,195,205] or by sending messages.[39,198,199]

The local load balancing can be divided into two separate steps. The first step determines how much work to send, and to which processor, and the second step determines which object will be migrated.

The first step is usually solved by employing a diffusion model[30] or some of its variations (demand-driven model,[39,198,199,196] dimensional exchange[30,198,206] and other[33,66,68,69,80,162,165]). The second step is usually solved by an algorithm which was first presented by Kernighan and Lin (KL algorithm)[88] or a modification of it.[189,169] There is a wide range of variations of both the diffusion model and the KL algorithm and it is out of the scope of this chapter to list and describe them all. An excellent review was presented by Hendrickson and Devine.[61]

Graph partitioners. In the graph model of computation, a task is represented in the graph as a vertex. If one task needs data owned by the other task (data dependency), the corresponding vertices are connected by an edge. The amount of computation is represented by a weight and the weight can be associated with both vertices and edges. Graph partitioners are usually employed to divide the computational mesh. These methods are mostly static and the partitioning is performed as a separate preprocessing task before the simulation. It follows that these methods are not very suitable for parallelisation of any method of discontinua. Nevertheless, attempts have been made to employ these for parallelisation of MD,[202] coupled DEM/CFD simulation[79]or FDEM.[138] Some popular graph partitioners are described below.

Recursive Spectral Bisection (RSB) method was first presented by Simon[172] and is based on a graph bisection partitioner introduced by Pothen et al.[150] It calculates an eigenvector of a Laplacian matrix of the graph. The values in the eigenvector give information about the weight of vertices in the graph and the graph can be partitioned if the vertices are sorted by their weight. This method is very expensive compared with geometric methods but the partitions are of a high quality. RSB method is not incremental.

Multilevel methods[17, 62, 85] are very popular partitioning algorithms. In the first step the graph is coarsened by joining vertices together in order to construct a smaller graph. This step is repeated until a sufficiently small graph is obtained. In the second step the smallest graph is partitioned. The third step involves a multilevel refinement which involves going back through each step until the original graph is reached. In each step the partition is refined. ParMETIS[84, 86, 169] and JOSTLE[190, 188] can serve as examples of parallel implementations of multilevel methods.

2.5.3 Parallel Processing of the Combined Finite-Discrete Element Method

The typical FDEM analysis combines a finite element-based analysis of continua with a discrete element-based transient dynamics, contact detection and contact interaction. While the parallelisation of both Finite Element Method (FEM) and Discrete Element method is covered in many published papers and books, the parallelisation of FDEM is somewhat less explored. The parallelisation solutions for FEM and DEM have a fundamentally different approach to the problem (while FEM in most cases does not need dynamic domain decomposition and load balancing, DEM does) and thus each of them cannot be directly utilised for the parallelisation of FDEM, which combines

features of both methods. An overview of the parallelisation efforts in DEM and related methods of discontinua is presented in Chapter 2.3. It is divided into sub-chapters corresponding with the parallel architecture used in each implementation.

Dynamic domain decomposition strategy for finite-discrete element simulation for shared-memory parallel computers was presented by Owen and Feng[138,139]. The finite element computation and discrete element computation was dynamically partitioned independently in the parallel implementation. A partitioner termed ParMETIS[86] (a MPI implementation of one of the most commonly used topological partitioners called METIS[83]) has been employed. Parallelisation for distributed-memory parallel computers following multiple-instructions/multiple-data (MIMD) paradigm was done by Wang et al.[192,191]. A master/slave approach was adopted in the above parallel implementations which means one master processor handles domain decomposition and load balancing tasks and distributes work to slave processors.

Munjiza et al.[122] presented some general strategies for parallelisation of FDEM. A hardware independent FDEM parallelisation framework by using Virtual Parallel Machine (PVM) and a static partitioner has been presented by Lei et al.[95] Schiava D'Albano[167] developed a static domain decomposition based parallelisation of FDEM by using Message-Passing Interface (MPI).

Since the use of the GPU architecture for numerical simulations is a relatively new idea, the parallelisation efforts of FDEM utilizing GPUs is limited. The GPU implementation of FDEM which is based on an open source FDEM code Y2D has been presented by Zhang et al.[209] The GPU parallelisation of coupled FEM/DEM approach (CDEM) has been presented by Wang et al.[193] and Ma et al.[106]

2.5.3.1 A Novel FDEM parallelisation strategy

The parallelisation strategy presented in this thesis builds on a master/slave approach developed for distributed-memory systems by Wang et al.[192,191]. Unlike Wang et.al.[192,191] all tasks (domain decomposition, load balancing, etc.) are performed concurrently on all processors. Parallelisation library chosen is Message-Passing Interface (MPI)[141,58]. The work presented is aimed at distributed-memory systems but can be used also on the shared-memory system.

The parallelisation strategy is designed around a dynamic domain decomposition approach using a modified Recursive Coordinate Bisection (RCB) algorithm.[174] Combined with the fact that all tasks are performed concurrently on all processors, this requires the design of the parallel implementation to be quite different from previous

parallelisation efforts. The input/output operations, the classification of finite elements (triangular elements), contact detection and contact interaction are re-designed so that they can be used together with the chosen partitioner and a completely new approach for the classification of joint elements is presented.

Since the joint elements are not supplied in the input file a novel parallel mesh engine is developed. This engine is for the purpose of generating joint elements on all processors concurrently during the run of the parallel implementation and as well as resolving the classification of each joint element according to its position in the sub-domain. Another function of the mesh engine is to turn the finite element mesh into a collection of discrete elements (heap of particles).

The design of the calculation of nodal forces and their exchange around the borders of sub-domains prohibits the use of an optimized communication engine offered by the parallelisation library. Its use would result in incorrect values of nodal forces. Thus a completely new communication engine is presented. The communication engine is tailored for the unique geometric properties of the grid of sub-domains generated by the modified RCB algorithm. Numerical simulations show that the communication scales well with the increasing number of processors (see Chapters 4 and 5).

Any dynamic domain decomposition based parallelisation strategy requires the moving of elements between sub-domains as well as load balancing and subsequent re-distribution of elements between sub-domains. A new implementation of these functions is presented with the developed classification of triangular/joint elements and geometric properties of the grid of sub-domains in mind.

The complete design of the FDEM parallel implementation covering all areas mentioned above is described in detail in Chapter 3.

2.6 Performance of a Parallel Implementation

The main reason for parallelising a sequential program is to increase performance. Thus, it is necessary to introduce some metrics by which the performance of a parallel implementation can be measured. Speedup and efficiency are the most commonly used performance metrics in parallel computing.

Speedup. The speedup is the ratio between the execution time of the sequential program t_s and the execution time of its parallel implementation on p processors t_p

$$S = \frac{t_s}{t_p} \tag{2.5}$$

According to some authors, the execution time of the sequential program should be for the best sequential algorithm run on the fastest computer.[142] In practice t_s is measured for a sequential program on which parallel implementation is based and run on one processor of the parallel machine. In case of the heterogeneous HPC cluster where each node contains multicore processors, the sequential program is run on a single core.

The execution time of the parallel implementation depends on the number of processors p used. In the case when:

$$t_p = \frac{t_s}{p} \tag{2.6}$$

the speedup is called linear. In other words linear speedup is when the speedup S is equal to the number of processors p used. In practice speedup is usually smaller. If we substitute equation 2.6 into equation 2.5 then the speedup is:

$$S \leqq \frac{t_s}{t_s/p} = p \tag{2.7}$$

It should be noted that superlinear speedup, when $S > p$ is also possible but it is, in most cases, caused by the extra amount of memory available on the parallel system. Another reason for this behaviour is when the implementation of the sequential program is not optimal.[197]

As noted above, the speedup is usually smaller than the number of processors p used. This is an unavoidable consequence of parallelising a sequential program. Running a program on multiple processors always introduces some overhead. For instance, in the case of a distributed-memory system, the overhead includes communication between nodes over the interconnecting network. The speed of data transfer is usually much slower than the local memory access. Thus the communication overhead is expected to be smaller for shared-memory systems. Another overhead is in the case a domain decomposition is employed to parallelise FDEM code, since it is necessary to check the position of objects comprising the simulation and, if necessary, migrate objects from one processor to another.

The total execution time t_p on p processors can be expressed as:

$$t_p = \frac{t_s}{p} + t_o \tag{2.8}$$

where t_o is a total parallel overhead which includes communication, migration of objects, etc.

The speedup obtained on p processors is generally expected to improve with increasing the problem size, since the amount of computation increases while the parallel overhead stays the same.

Efficiency. The efficiency E of the parallel implementation is defined as a ratio between speedup S and the number of processors p. If the equation 2.5 is substituted then the efficiency can be expressed as follows:

$$E = \frac{S}{p} = \frac{t_s}{p \cdot t_p} \tag{2.9}$$

Amdahl's law. It should be noted that the sequential program cannot often be parallelised as a whole. If the fraction which cannot be parallelised is f then the total computational time on p processors will be $ft_s + (1-f)t_s/p$. The part ft_s is time spent calculating the sequential fraction of the program and $(1-f)t_s/p$ is the execution time of the parallelised fraction of the program. By substituting the total computational time into equation 2.5 the speedup can be expressed as:

$$S = \frac{t_s}{ft_s + (1-f)t_s/p} \tag{2.10}$$

After eliminating t_s the speedup is:

$$S = \frac{p}{f(p-1)+1} \tag{2.11}$$

The equation 2.11 is known as *Amdahl's law*.[5] Put into words, it means that the maximum obtained speedup, even for thousands of processors, is limited. The maximum speedup depends on the fraction f of the sequential code which could not be parallelised. Equation 2.11 can be rewritten into:

$$S = \frac{1}{f + (1-f)/p} \tag{2.12}$$

For $p \to \infty$ the speedup is:

$$\underset{p\to\infty}{S} = \frac{1}{f} \tag{2.13}$$

If just one percent of the code cannot be parallelised, then the maximum speedup calculated from equation 2.13 will be equal to 100 even if the code is run on thousands of processors. In practice this is not true. All the equations above assume that the speedup is only a function of the number of processors $S(p)$. In fact the speedup $S(p,n)$ is a function of both the number of processors p and the number of objects

(problem size) in the simulation n. Thus, with increasing the problem size, the fraction f decreases and the speedup grows.

Scalability. In context of the parallel program performance the parallel program is scalable, if the efficiency E (eq. 2.9) remains constant while both the problem size and the number of processors increase at the same rate.[142] The case, when efficiency E remains constant while the number of processors is increasing and the problem size is fixed, is called *strong scaling*. The *weak scaling*, on the other hand, occurs in the case of the efficiency E remaining constant, while both the number of processors and the problem size increase at the same rate.

2.7 Floating Point Arithmetic

Many real numbers cannot be stored in the memory of the computer exactly. Due to the limited amount of memory, each is stored as a floating point number, which is only an approximate representation of the real number. Also, mathematical operations on floating point numbers produce results too big to be stored whole in the memory and the result must be rounded in order to get its finite representation. This introduces a so-called *rounding error* which is an unavoidable consequence of the floating point computation.

Any floating point number is saved in the computer's memory in the following format:[52]

$$\pm d_0.d_1d_2\cdots d_p \times \beta^e \tag{2.14}$$

where $d_0.d_1d_2\cdots d_p$ is a significand with precision p, β is a base and e is an exponent.

The real number is rounded to the closest floating point number. The *relative error* of this operation can be calculated as follows:

$$relative\,error = \frac{real\,number - floating\,point\,representation}{real\,number} \tag{2.15}$$

When the real number is rounded to the floating point number, the maximum relative error of this operation is bounded by a so-called *machine epsilon* ε. The values of machine epsilon are prescribed by IEEE standard and they depend on the precision p, base β and the number of bits allocated in the memory (in C++ data type float, double).

The presence of the rounding error has one very important consequence for the scientific computation executed in parallel: The results obtained for exactly the same input file executed on a different number of processors p will vary from each other

and it follows that the result calculated by using sequential version of the same code will be different as well. This is because of the fact that, for instance, a summation of contact forces (in DEM, FDEM simulation) will be performed for different numbers of processors p in a different order, which introduces an error in summation.

The error in summation can be shown by a simple algorithm. If we create an array of one million numbers of type double and fill it with numbers $0.1, 1.1, \cdots, 999999.1$ and then perform a summation of all numbers in ascending order and then in descending order, the results will be:

$$\begin{aligned} sum\,ascending = &\quad 499999599995.69922 \\ sum\,descending = &\quad 499999599985.11359 \end{aligned} \tag{2.16}$$

It can be seen from the results in equation 2.16 that the difference is around 10.5. The starting number 0.1 was chosen because it cannot be saved in the memory accurately because of the conversion to the binary format. Thus, it is represented only by a floating point approximation. The results in equation 2.16 were obtained on a computer with 64-bit operating system.

The approximate representation of real numbers can also lead to the sensitivity to initial conditions and the loss of symmetry in the discontinua simulation as pointed out by Munjiza.[121]

It is out of the scope of this thesis to provide details about all aspects of floating point arithmetic. An excellent review was presented by Goldberg.[52]

Chapter 3

NOVEL SPACE DECOMPOSITION BASED PARALLEL SOLUTIONS FOR THE COMBINED FINITE-DISCRETE ELEMENT METHOD

3.1 Introduction

This chapter provides a complete description of both the design and the implementation of the proposed parallel solutions. All main areas are covered in detail, including the classification of triangular/joint elements, contact detection and contact interaction, migration of elements between processors, load balancing and subsequent re-distribution of elements, as well as the proposed communication engine.

A geometric partitioner based on Recursive Coordinate Bisection (RCB) algorithm, first introduced by Berger and Bokhari,[11] has been adopted in this work because shapes of sub-domains generated by ParMETIS and other graph partitioners are irregular and, therefore, add to the complexity of a parallel implementation.

In order to demonstrate the parallelisation strategy developed in this thesis, the computational domain is partitioned into a rectangular grid of sub-domains generated by a modified RCB algorithm.[174] Each sub-domain is assigned to a single processor in the PC cluster. The parallelisation strategy aims at performing all tasks (domain decomposition, load balancing) concurrently on all processors.

Sequential FDEM program (Y2D code) is written in C, and Message-Passing Interface (MPI)[141,58] has been chosen as the parallelisation library. MPI is one of the

most commonly used libraries for parallel programming. Message passing is used to exchange data between processors meaning processors coordinate their activities by explicitly sending and receiving messages. A short overview of MPI is given in Chapter 2.4.2. It should be noted that each MPI process is assigned to one processor, so words process, processor and sub-domain can be interchangeably used.

3.2 Layout of a Sequential FDEM code

A single processor, in-house, finite-discrete element program called Y2D was originally designed for the purpose of demonstrating some concepts described in Munjiza's "The Combined Finite Discrete Element Method".[117]

The equations of motion of the finite-discrete element system is solved by a time integration scheme, based on a central difference method, in which FDEM code performs at each time step:

- Calculation of internal forces:
  - forces calculated from deformation of finite elements,
  - forces calculated from deformation (opening, sliding) of joint elements.
- Application of external loads.
- Contact detection.
- Contact interaction (contact forces).
- Solution of equations of motion for each element separately.
- Writing output (only in time steps specified in input file).

Details on each computational procedure and methods used can be found in Munjiza's FDEM book.[117] The whole sequential Y2D code is summarized in Figure 3.1.

3.3 Layout of a Parallel FDEM code

In order to implement a parallel version of the Y2D code, some additional calculations must be performed. Firstly, forces of elements located on boundaries between sub-domains must be exchanged in each time step.

Secondly, since a typical FDEM problem is dynamic in nature, it is necessary to migrate elements between sub-domains during the run of the simulation. If a work-load imbalance is too big, re-partitioning and load balancing must be performed. A flowchart of the parallel Y2D code is in Figure 3.2. Coloured boxes in Figure 3.2 represent newly created parallel functions as well as originally sequential functions, but with major changes in order to work in parallel. Design and implementation of each function is described in detail in the following chapters.

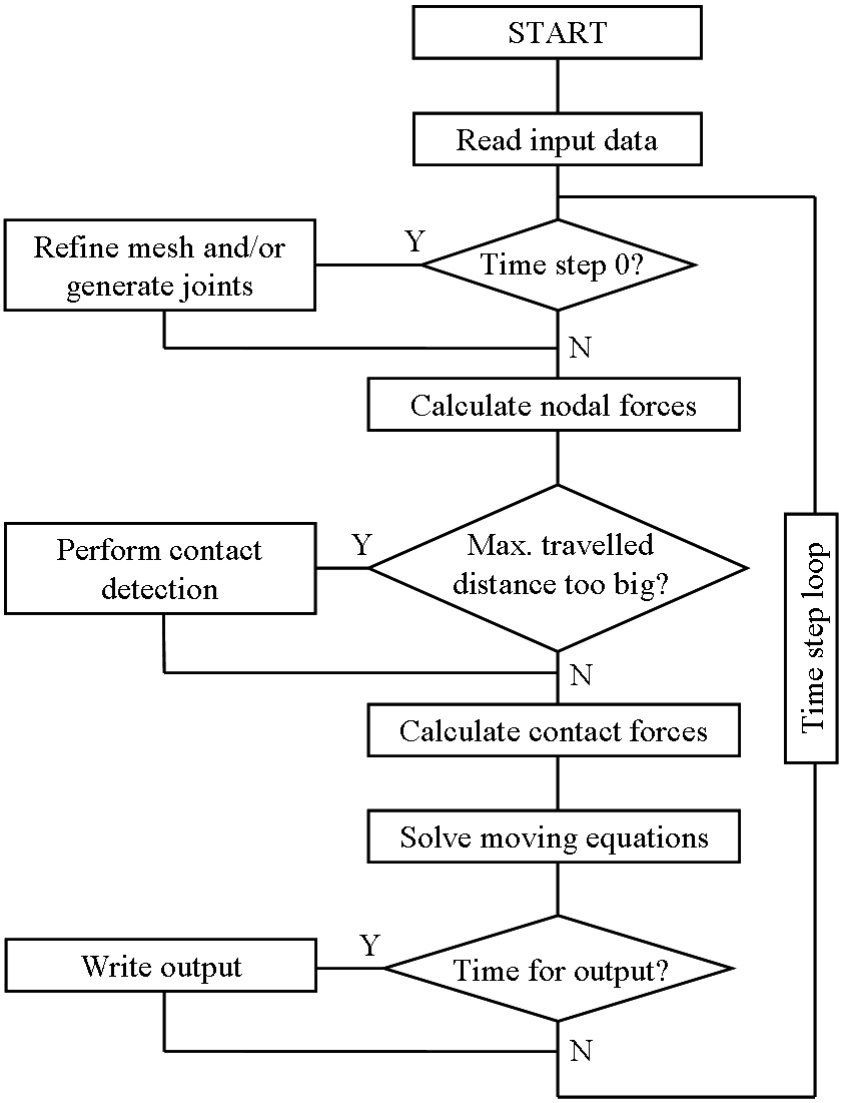


Figure 3.1: Flowchart of a sequential Y2D code.

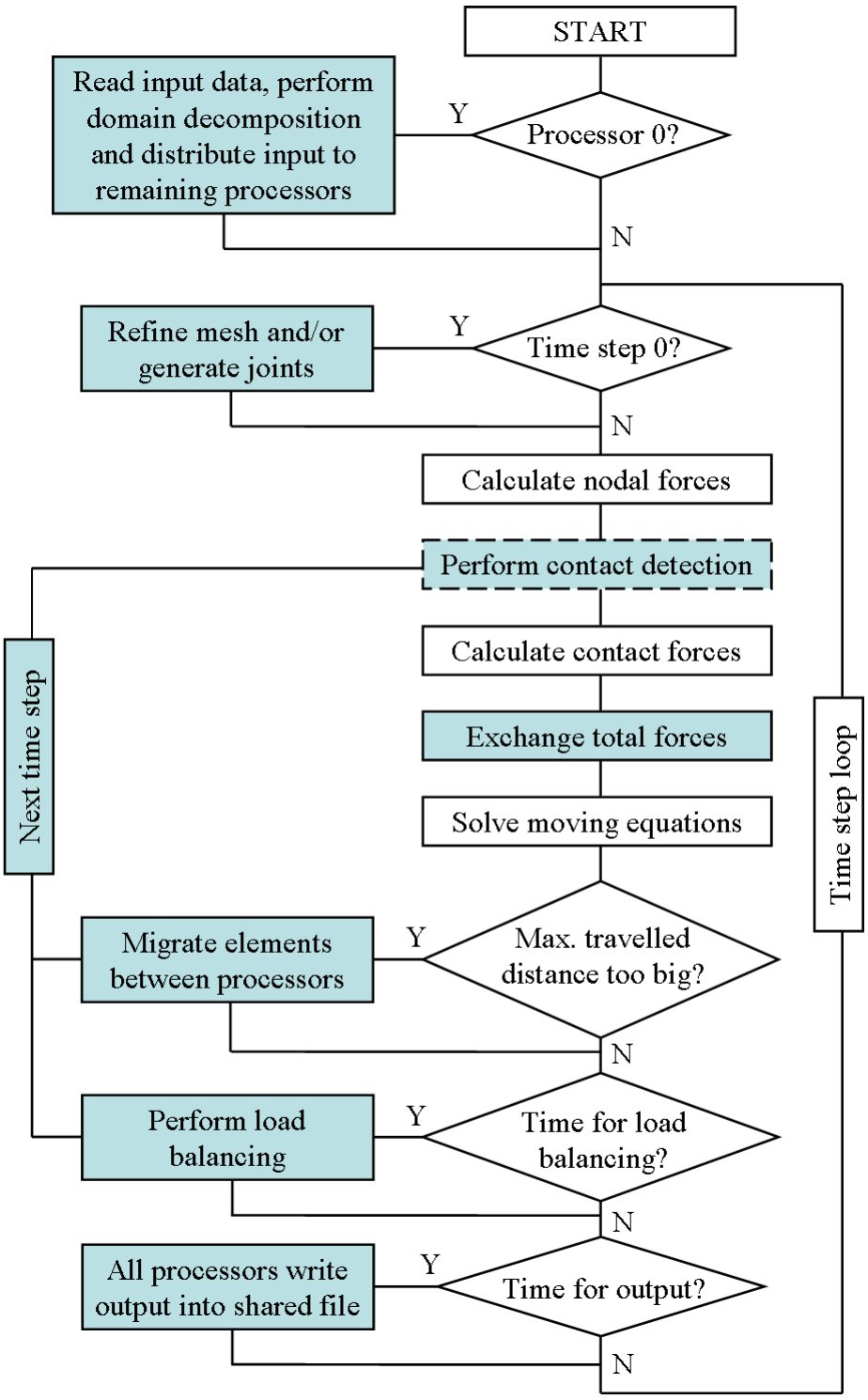


Figure 3.2: Flowchart of a parallel Y2D code.

3.4 Classification of Elements Depending on their Location within Sub-domain

3.4.1 Buffer Zone around Borders of Sub-domain

The shape of each sub-domain is set to a rectangle in order to adopt a chosen partitioning algorithm. A buffer zone is introduced around the borders of each sub-domain, see Figure 3.3. The size of each sub-domain is therefore expanded by half the size of the buffer zone and the sub-domains are overlapping the borders. Elements located in the overlapping region are shared among the processors these sub-domains were assigned to.

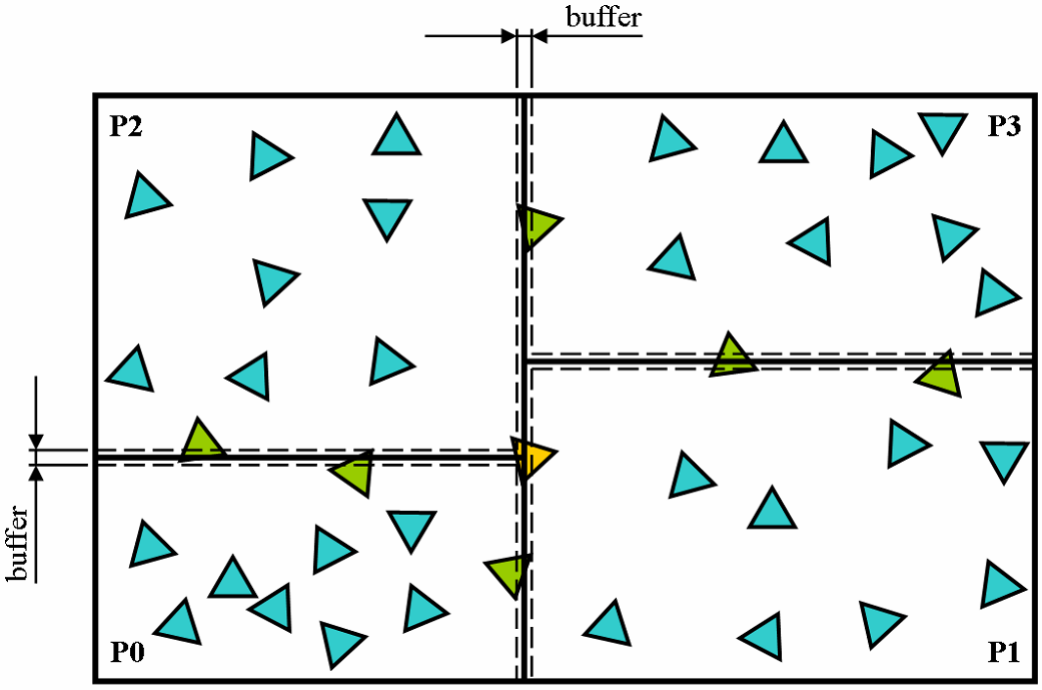


Figure 3.3: A buffer-zone introduced around borders of sub-domain.

The buffer zone is created for the purpose of controlling the frequency of domain decomposition (migration of elements from one processor to another) which is a very expensive operation in terms of CPU time. Hence the elements are not migrated between each processor in every time step but only if the maximum distance travelled exceeds the specified value.

The size of the buffer zone B_{DD} is closely tied to the size of the buffer controlling the frequency of contact detection B_{CD}, since the migration of elements requires new

contact detection search to be performed. The size of the buffer zone B_{DD} is calculated as follows:

$$B_{DD} = B_{CD} \cdot I_{DD} \tag{3.1}$$

where I_{DD} is a parameter bigger or equal to 1.

The size of the buffer zone has a big impact on the performance of the parallel implementation. A larger size means higher numbers of elements in the sub-domain which means an increase of computation and communication for each time step. On the other hand, the frequency of domain decomposition will decrease. A smaller buffer zone means there are less elements in the sub-domain therefore, a decrease in computation and communication for each time step, but the domain decomposition will be performed more often, increasing the cost of domain decomposition. Consequently, the size of the buffer must be chosen based on numerical experiments to achieve the best overall performance of the parallel implementation, see Chapter 4.

3.4.2 Status of Constant Strain Triangular Element

Since domain decomposition is performed using a geometric approach, all elements are divided into several categories depending on their location within the sub-domain. If a constant strain triangular element is located totally inside the sub-domain, it is marked as internal (A), see Figure 3.4. An element overlapping with the buffer zone of sub-domain is marked as interfacial. Thus, the status of the element can be either internal or interfacial. Depending on the location of an element, interfacial elements can be divided into three categories (Figure 3.4):

- Element located at the border between two sub-domains is shared among two processors and marked as interfacial (B).
- If an element is located at the corner and shared among three processors, it is marked as interfacial (C3). This rule actually applies only to two out of three processors. For the third processor the element is located at right/left border of sub-domain and top/bottom borders of neighbouring two processors, see Figure 3.4.
- If an element is located at the corner and shared among four processors, it is marked as interfacial (C4).

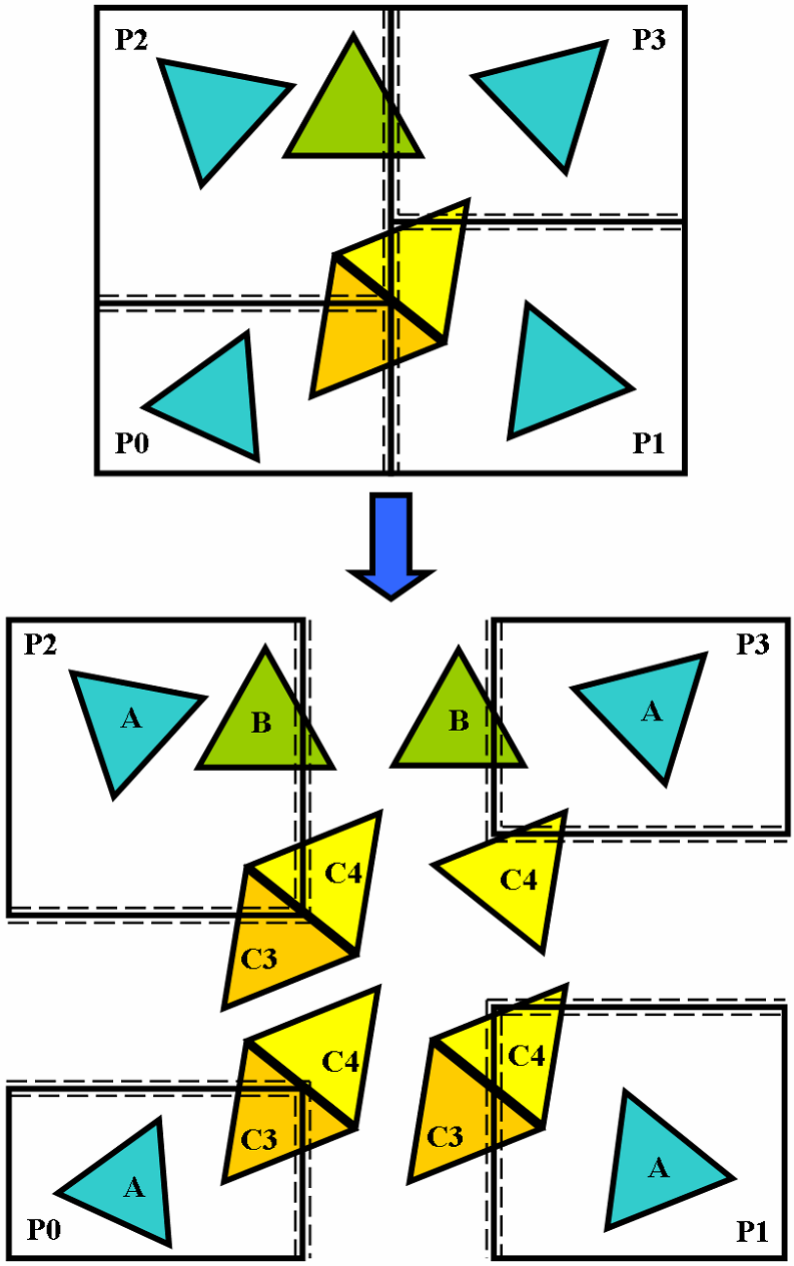


Figure 3.4: A status of a constant strain triangular element.

3.4.3 Status of Joint Elements

The joint element in the FDEM code is a four-node element functioning as a bond between triangular elements. The joint element represents a fracture mechanism called "discrete crack model" developed by Munjiza[117] in order to introduce a transition from continua to discontinua in the Combined Finite-Discrete Element Method.

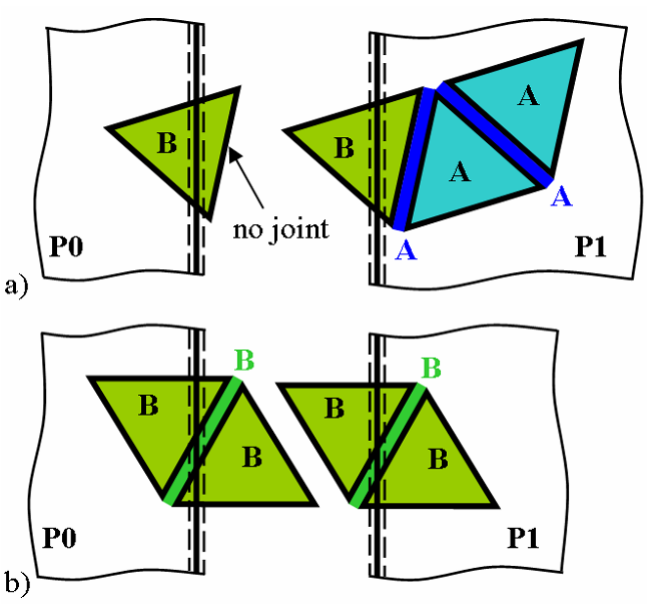


Figure 3.5: A status of a joint element: a) a combination of A-A/B, b) a combination of B-B.

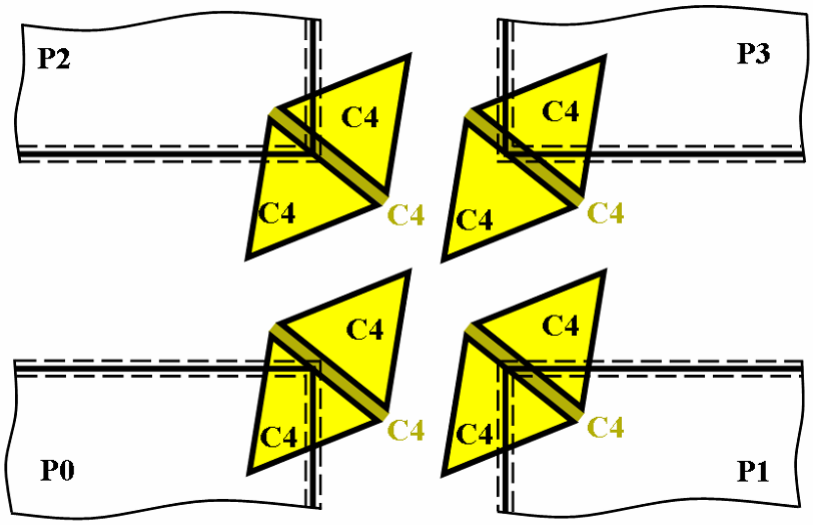


Figure 3.6: A status of a joint element for a combination of C4-C4.

The status of the joint element is not based on the location within the sub-domain but rather on the combination statuses of the two triangular elements it is attached to:

- An internal triangular element (A) in combination with any other status gives internal joint element (A), see Figure 3.5a.

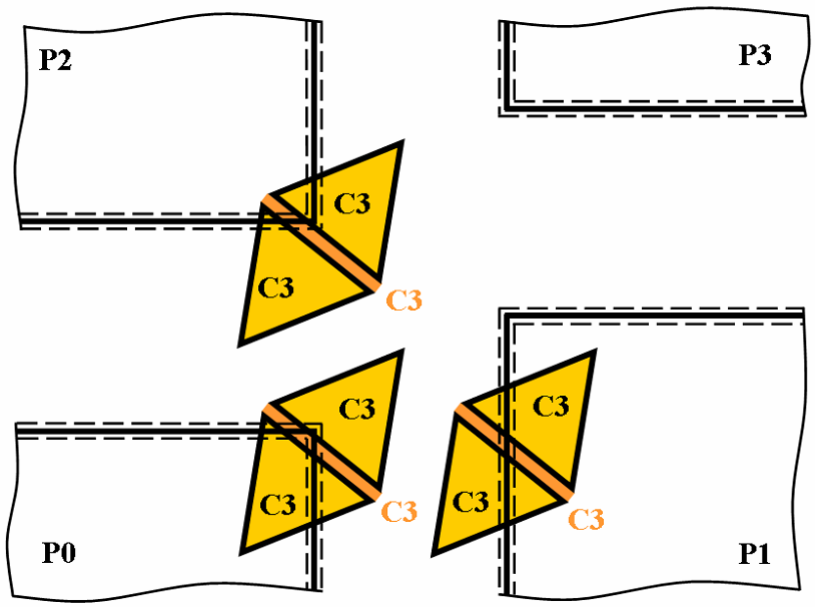


Figure 3.7: A status of a joint element for a combination of C3-C3.

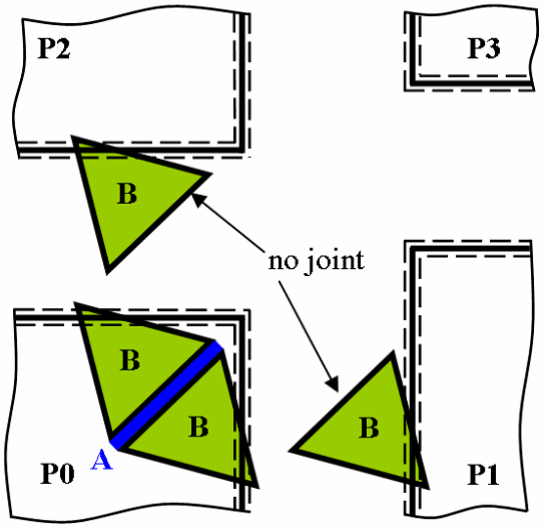


Figure 3.8: A status of a joint element for a combination of B-B located at perpendicular borders.

- Combinations of B-B, C4-C4 and C3-C3 give interfacial joint element B (Figure 3.5b), C4 (Figure 3.6) and C3 (Figure 3.7) respectively with two exceptions:

  – For the combination of B-B if one triangular element is located at the horizontal border and the second one is located at the vertical border, the joint element must be marked at internal (A), see Figure 3.8.

– For the combination of C3-C3 if one triangular element is located at the corner and the second one is located on the same processor at the right/left border, the status of the joint element must be interfacial (B) since the joint is shared only by two processors out of four, see Figure 3.9.

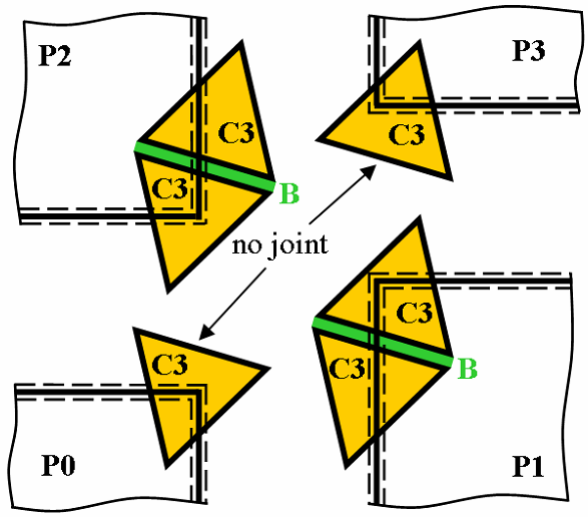


Figure 3.9: A status of a joint element for a combination of C3-C3 located at corner-border.

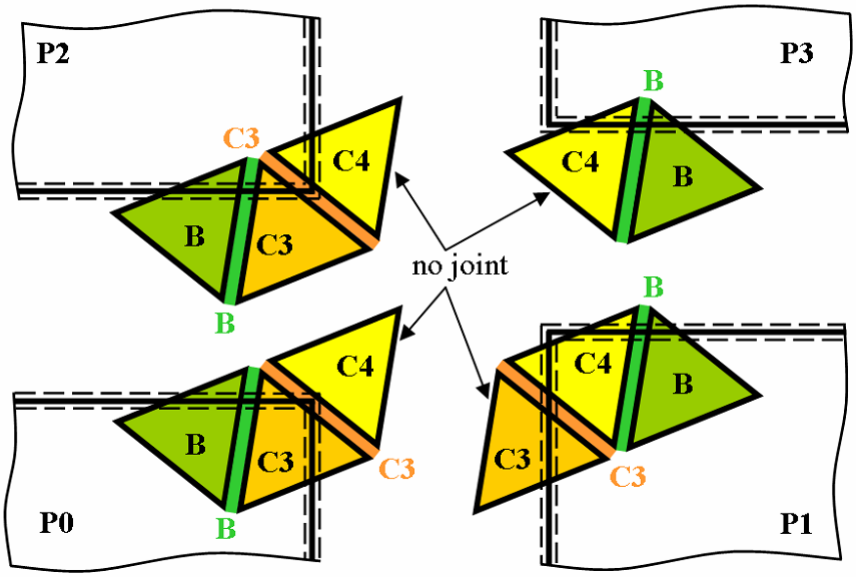


Figure 3.10: A status of a joint element for combinations of B-C3/C4 and C3-C4.

- A combination of B-C3/C4 gives a status of joint element as B (Figure 3.10) since both triangular elements are located only on two processors and only one

triangular element is present on the remaining processor(s).

- A combination of C3-C4 gives interfacial joint element C3 since interfacial triangular element is shared among three processors, see Figure 3.10.
- Joint element attached to only one triangular element is deleted, see Figures 3.5-3.10.

3.5 Domain Decomposition

Since a typical problem in FDEM dynamically changes in an unpredictable way during the run of the simulation, a dynamic partitioner must be chosen to perform domain decomposition. A modified RCB algorithm[174] is used to decompose the computational domain.

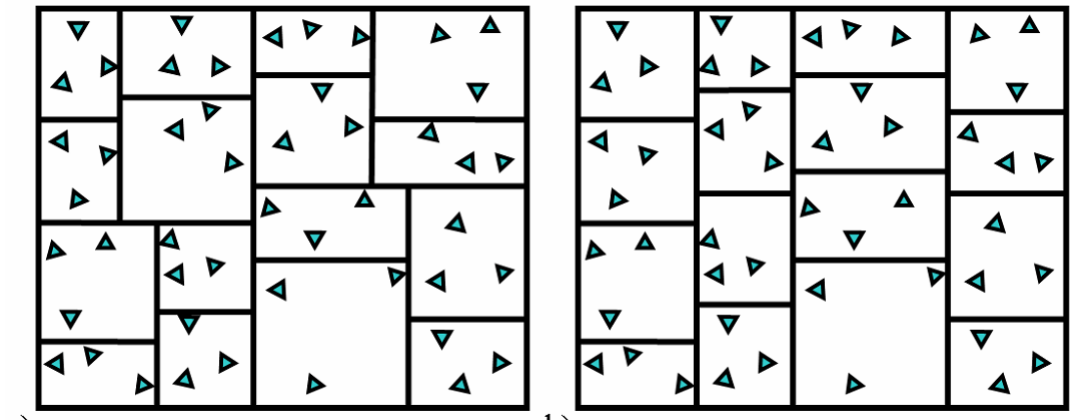


Figure 3.11: Partitioning to 16 sub-domains performed a) non-hierarchically, b) hierarchically.

An advantage of the modified RCB algorithm is a significant reduction in the complexity of programming since number of neighbouring processors at each horizontal border is fixed to one, see Figure 3.11. The modified RCB algorithm is partitioning the computational domain hierarchically unlike the original RCB algorithm which is partitioning the domain randomly. Hierarchical partitioning means the domain is systematically partitioned at different levels. The number of partition levels equals the dimensionality of the domain; two for a 2D problem.

Figure 3.11a shows a 2D grid of 16 sub-domains created by the original RCB algorithm and Figure 3.11b shows a 2D grid partitioned by the modified RCB algorithm. It

is evident that the modified RCB algorithm is partitioning the computational domain in two steps. In the first step the domain is divided in x direction into a specified number of columns each with equal load and, in second step, each column is again divided in y direction into a specified number of rows.

Algorithm 3.1 Partitioning in x direction.

```
 1: integer irow, icol;                                  ▷ Number of rows (columns).
 2: integer nelem;          ▷ Sum of all cells in the whole computational domain.
 3: Calculate nelem;
 4: xelem = nelem/icol;                                  ▷ Target load for each column.
 5: for (i = 0; i < (icol − 1); i++) do                  ▷ Last column uses global border.
 6:     Set psum, csum to zero;                          ▷ Previous (current) sum set to zero.
 7:     for (ix = xcell; ix < ncelxg; ix++) do           ▷ ncelxg is global number of x cells.
 8:         psum = csum;
 9:         csum = csum + sum of all ix cells;
10:         if csum ≧ xelem then                        ▷ Is greater or equal than target load.
11:             diff1 = csum − xelem;                   ▷ Difference at current cell.
12:             diff2 = psum − xelem;                   ▷ Difference at previous cell.
13:             if diff1 ≧ diff2 then
14:                 Save border i at previous cell (ix − 1);
15:                 xcell = ix;                         ▷ Set xcell for next column.
16:             else
17:                 Save border i at current cell (ix);
18:                 xcell = ix + 1;                     ▷ Set xcell for next column.
19:             end if
20:             break;
21:         end if
22:     end for
23: end for
```

In order to determine the load during partitioning, the computational domain is discretized into fine cells and the load in each cell is determined. In the case of first domain decomposition at the start of the simulation the load is given by the number of elements in each cell, as the partitioning is finished before the first contact detection search is performed. Estimation of load in each cell during load balancing is described in detail in Chapter 3.11.

The whole partitioning algorithm can be outlined as follows:

1. Discretize computational domain into cells and determine load in each cell, see Chapter 3.11 for details.

2. Partition the domain in x direction into specified number o columns, see Algo-

rithm 3.1.

3. Loop over all created columns and partition each into specified number of rows by using Algorithm 3.1 again (coordinate x in the algorithm is replaced by y coordinate and *nelem* is replaced by the sum of all cells in the corresponding column).

Domain decomposition is done at the start of the simulation and then every time load balancing is triggered. After the first domain decomposition is finished, the processor with rank 0 checks the position of elements (according to rules described in Chapter 3.3) and distributes the input data to remaining processors. For details on load balancing see Chapter 3.11.

3.6 Parallelisation of Meshing

3.6.1 Introduction

The mesh function in sequential Y2D code includes a couple of algorithms with different capabilities. It is essential to parallelize at least two of those algorithms in order to fully utilize FDEM.

The first algorithm generates joint elements for a given finite element mesh and therefore enables fracture and fragmentation. This algorithm is a necessary part of Y2D code since no external mesh generator is available to accomplish this. Figure 3.12 shows an example of a joint element generated between two triangular elements. Joints are also generated on remaining edges as well but these joints belong only to one triangular element.

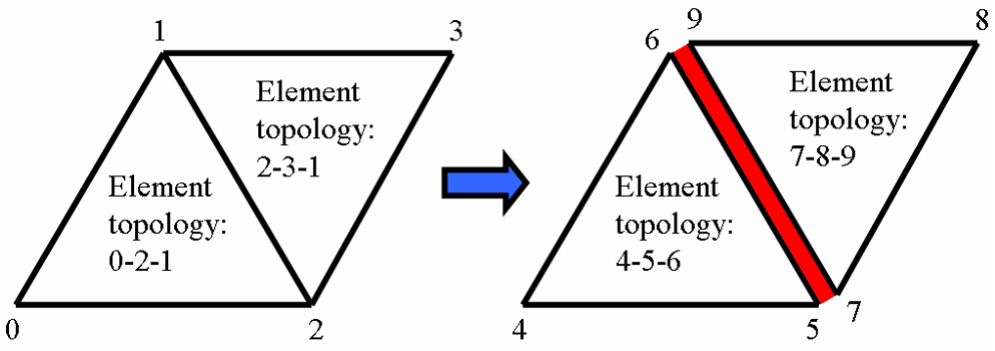


Figure 3.12: Joint element generated between two triangular elements.

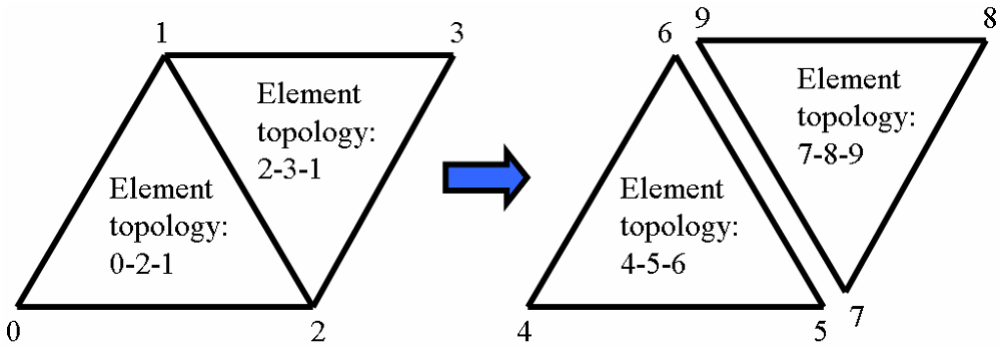


Figure 3.13: Discrete elements generated from two finite elements.

The second algorithm separates nodes from each other and, hence, turns finite elements into a heap of separate particles, i.e. discrete elements, see Figure 3.13.

It is evident from Figures 3.12 and 3.13 that both algorithms are very similar. Copies of original nodes are created in both cases and created nodes are connected by joint elements in the first case.

3.6.2 Global Numbers of Nodes

In order to make movement of elements and nodes possible, a global ID (GID) of triangular elements and nodes must be introduced. This is done on the processor with rank 0 during domain decomposition at the start of the simulation. These global numbers are then distributed to remaining processors with the rest of the input data and stored in the database.

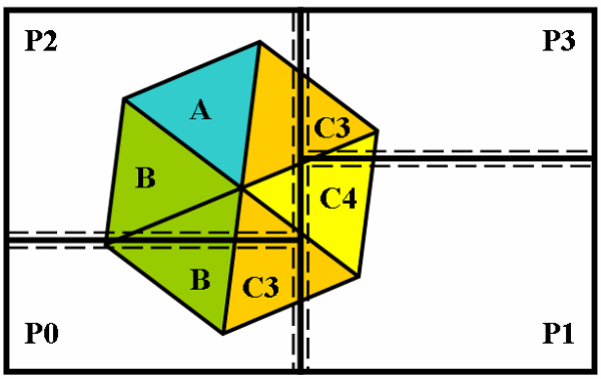


Figure 3.14: Elements shared by four processors to be meshed in parallel.

If either meshing algorithm is called, new nodes are created and each new node must be assigned a unique global ID. Moreover nodes created by the meshing of in-

terfacial elements are present on more than one processor and it is necessary to ensure that the same node is assigned the same global ID on all the processors it belongs to.

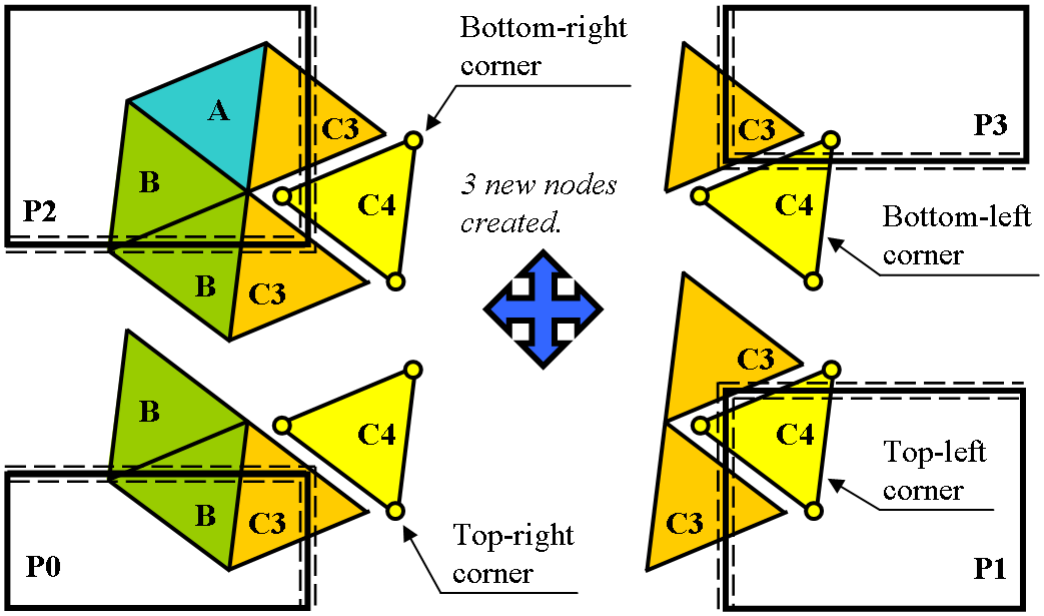


Figure 3.15: Nodes created by meshing of interfacial elements C4.

To resolve an assignment of global IDs of nodes properly, triangular elements cannot be meshed in random order. Interfacial and internal elements on each processor (see Figure 3.14) must be processed during meshing in the following order:

- Interfacial elements C4 located at top-right, top-left, bottom-left and bottom-right corner respectively, see Figure 3.15.
- Interfacial elements C3 located at top-right, top-left, bottom-left and bottom-right corner respectively, see Figure 3.16.
- Interfacial elements C3 located at right and left border respectively (each neighbouring border separately), see Figure 3.16.
- Interfacial elements B located at: a) right border and each segment (neighbouring processor) separately in the ascending order, b) top border, c) left border and each segment separately in the ascending order, d) bottom border (Figure 3.17).
- All internal elements A (Figure 3.17).

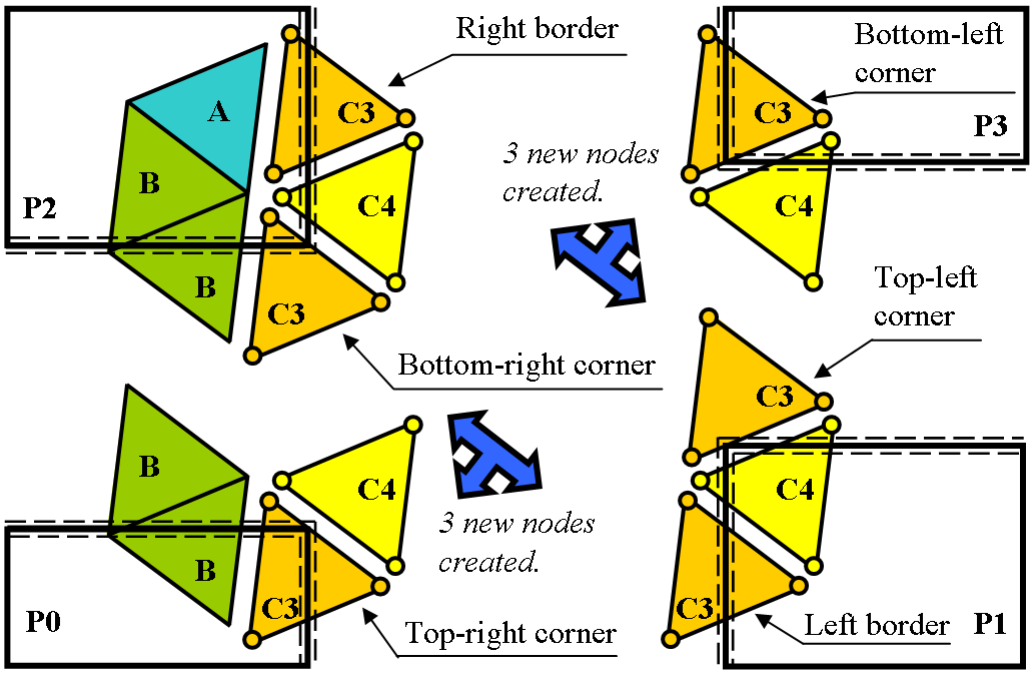


Figure 3.16: Nodes created by meshing of interfacial elements C3.

Each status is meshed separately and the count of new nodes is saved. As a result, the count of new nodes for the corresponding border/corner is exactly the same on corresponding processors. For instance the count of new nodes (3 nodes) at top-right corner on processor with rank 0 is the same as count at top-left, bottom-left and bottom-right corners on processors 1, 2 and 3 respectively, see Figure 3.15. Moreover these nodes were created in the exactly same order on all four processors. This enables each processor to assign correct global IDs independently from the remaining processors.

The next step after meshing is to gather counts of new nodes on each processor. For that, a MPI function MPI_Allgather() is used. It functions as a collective communicator, gathering messages from each processor and distributing them to all processors. For instance, if each processor in Figures 3.14-3.17 sends in the message its rank, as a result of calling MPI_Allgather(), all processors would receive an array of four integer numbers 0, 1, 2 and 3.

Each processor is required to assign unique global IDs of new nodes created by meshing the following statuses only and in the following order:

- Interfacial elements C4 located at top-right corner.
- Interfacial elements C3 located at top-right corner.

- Interfacial elements C3 located at right border (for each neighbouring border separately), see Figure 3.18.

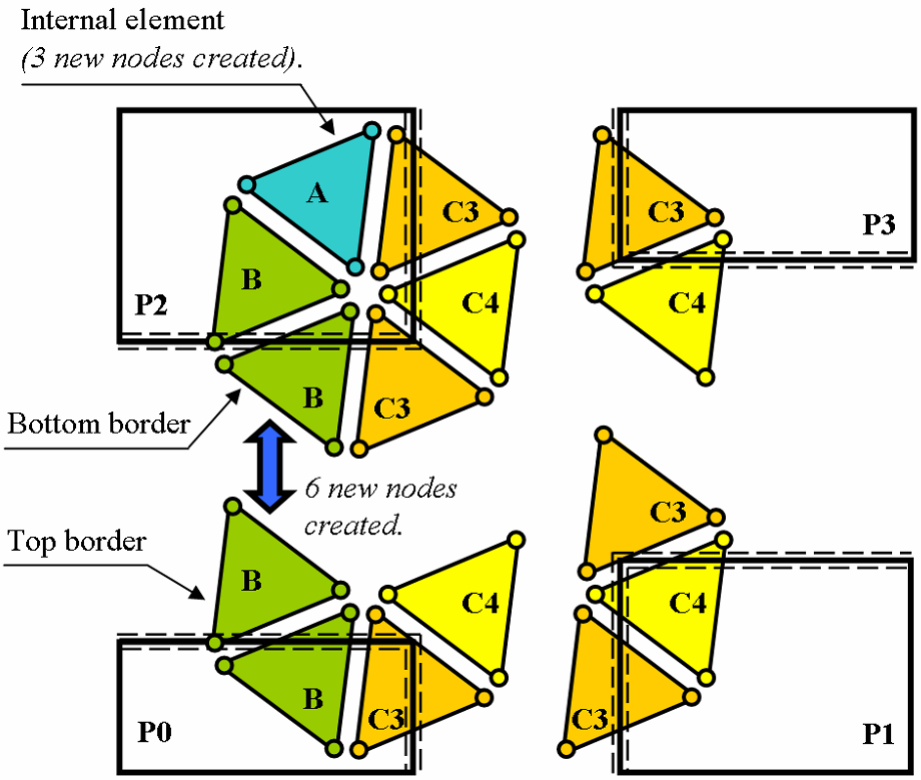


Figure 3.17: Nodes created by meshing of internal elements A and interfacial elements B (processors 0 and 2 only).

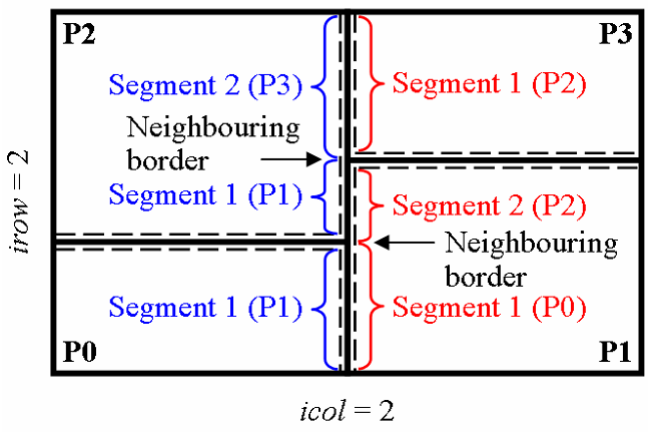


Figure 3.18: Segments (neighbouring processors) at right/left border for each processor.

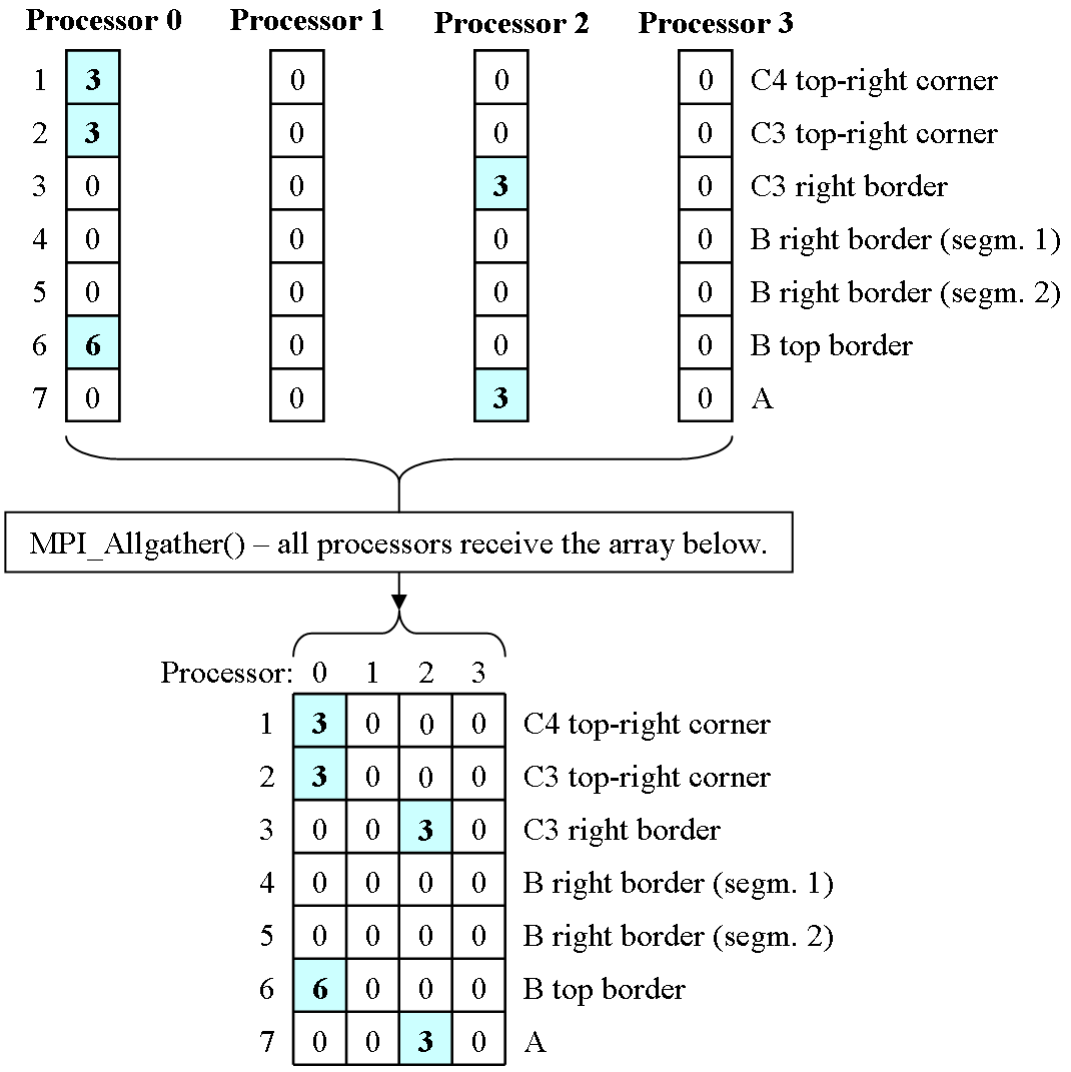


Figure 3.19: Gather operation on counts of new nodes saved on each processor.

- Interfacial elements B located at right border and each segment (neighbouring processor) separately in the ascending order, see Figure 3.18.
- Interfacial elements B located at top border (processor has only one neighbouring processor at top border and therefore one segment only).
- All internal elements A.

Counts of new nodes belonging to the above statuses combined with the global number of nodes before meshing gives enough information to assign global IDs of nodes on each processor correctly, hence counts from each processor are gathered by using MPI_Allgather(), see Figure 3.19.

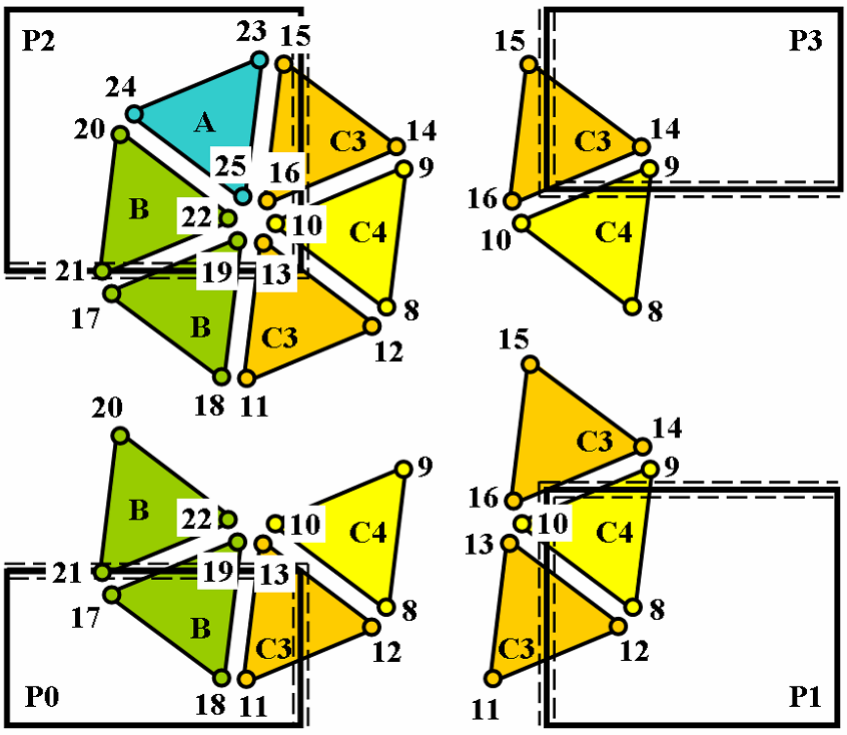


Figure 3.20: Global numbers of new nodes.

Global IDs of new nodes, created by meshing the above statuses, are assigned directly on each processor. That means that global IDs for the remaining statuses are assigned on neighbouring processors, nevertheless, the current processor is able to calculate what numbers were assigned on neighbouring processors. For example, for the numbering of nodes created by meshing of interfacial elements C4, see Figure 3.15:

- Processor 0: interfacial elements C4 are located at top-right corner hence numbers of new nodes are assigned on current processor. Sum of all counts *sump* - on processors with rank $irank < iproc$ (*iproc* is local processor) is calculated ($sump = 0$ since $iproc = 0$). Global number of nodes before meshing $nnopo = 7$ is the last information needed to assign IDs to new nodes. Then numbering of new nodes starts from $GID = nnopo + sump + 1 = 8$. Three new nodes created by meshing C4 are therefore assigned numbers 8, 9 and 10, see Figure 3.20.

- Processors 1-3: nodes created by meshing of C4 on these processors must be assigned the same numbers (8, 9, 10) as on processor 0. Firstly, the rank of neighbouring processor *nproc*, where the numbers of nodes for C4 at top-right corner are assigned, must be found. For instance, at the processor with rank

2, new nodes are located at the bottom-right corner and the corresponding top-right corner is located on processor 0, hence $nproc = iproc - icol$, see Figure 3.18. Secondly, the sum of counts of all previous statuses' *sums* in the array (Figure 3.19) on neighbouring processor ($nproc = 0$) is calculated ($sums = 0$). Since $nproc = 0$ sum of all counts *sump* on processors with lower rank than *nproc* is equal to 0. This means that processors 1-3 reproduce calculations done on processor 0 and therefore $GID = nnopo + sums + sump = 8$ and assigned numbers are 8, 9 and 10, see Figure 3.20.

Algorithm 3.2 Assignment of global IDs of new nodes.

```
1:  integer *i1nnew;          ▷ 1D array containing counts of new nodes for each status.
2:  integer **i2nnew;         ▷ 2D array containing gathered counts for each processor.
3:  integer nnopo;                          ▷ Global number of nodes before meshing.
4:  integer iproc,nproc;          ▷ Rank of local processor, neighbouring processor.
5:  Gather i1nnew into i2nnew;                  ▷ MPI_Allgather(), see Figure 3.19.
6:  for (istat = 0;istat < nstat;istat++) do      ▷ Loop over all element statuses.
7:      if istat = local then                ▷ If numbering decided on local processor.
8:          Calculate sump for irank < iproc;          ▷ Sum of statuses on 0 to irank.
9:          GID = nnopo + sump + 1;                    ▷ Start numbering from GID.
10:         for (i = 0;i < i1nnew[istat];i++) do ▷ Assign numbers (nodes for istat).
11:             global ID of node = GID;
12:             GID = GID + 1;
13:         end for
14:     else                          ▷ If numbering decided on neighbouring processor.
15:         Find rank of neighbouring processor nproc;
16:         Calculate sums on nproc;          ▷ Sum of statuses < istat on nproc.
17:         Calculate sump for irank < nproc;          ▷ Sum of statuses on 0 to irank.
18:         GID = nnopo + sums + sump + 1;             ▷ Start numbering from GID.
19:         for (i = 0;i < i1nnew[istat];i++) do ▷ Assign numbers (nodes for istat).
20:             global ID of node = GID;
21:             GID = GID + 1;
22:         end for
23:     end if
24: end for
25: nnopo = nnopo + sum of the whole i2nnew;      ▷ Update global number of nodes.
```

The remaining statuses, except new nodes of internal elements, are resolved analogically. One processor assigns global IDs of new nodes and corresponding neighbouring processors repeat the same calculations independently. Nodes of internal elements are only saved in one sub-domain, hence each processor assigns unique global

IDs to its internal nodes in the same way as to nodes of interfacial elements (calculate *sump* and *GID*). Resulting global numbers of all new nodes are in the Figure 3.20.

The whole process of assigning global IDs of new nodes is summarized in Algorithm 3.2. This algorithm can be used for both joints and discrete elements generation.

3.6.3 Mesh Generation of Joint Elements in Parallel

As mentioned above, generation of joint elements is very similar to discrete elements generation. Copies of nodes of triangular elements are created and in case of joint elements generation, these copies are assigned to a joint element, see Figure 3.12. The topology of the generated joint element from Figure 3.12 would be: 5-6-9-7. Each joint element is acting as a bond between two triangular elements, connecting an edge of the first triangular element to an edge of the second one.

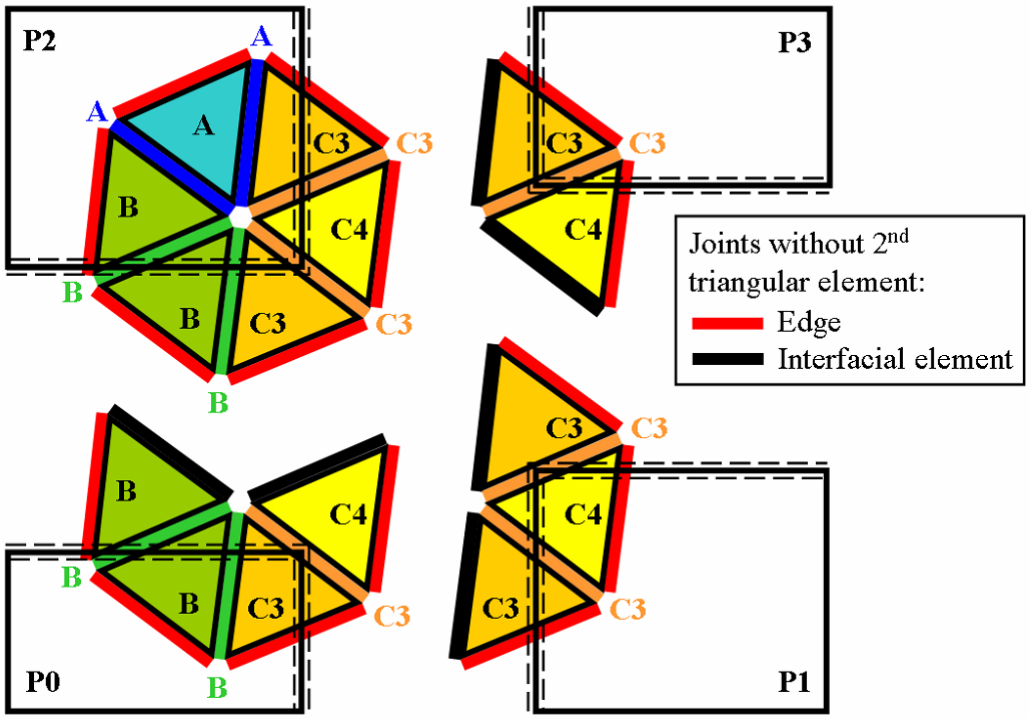


Figure 3.21: Joints generated without second triangular element.

Joint elements without a second triangular element are also generated during meshing. Two types of these elements can exist:

- Joints generated at the boundaries (edges) of the finite element mesh, see Figure 3.21.

- Joints generated for interfacial elements, for instance, if joint in first sub-domain belongs to internal (A) and interfacial (B) elements, then corresponding joint in second sub-domain is missing the first triangular element (internal element in first sub-domain), see Figure 3.21.

Joints generated for interfacial elements must be deleted according to the rules outlined in Chapter 3.4.3. Joints located on the boundaries can also be deleted since they do not act as a bond between triangular elements and, therefore, nodal forces calculated for these joints are equal to zero.

Triangular elements are not processed randomly during meshing but in a meshing order (described in the previous chapter) that correctly assigns global IDs to the newly created nodes. During meshing pointers to each joint's two triangular elements are saved into the element database, since this information is readily available.

When the meshing is done, each joint is assigned a status dependent on the combination of statuses of its triangular elements (see Chapter 3.4.3) and joints without a pointer to a second triangular element are deleted. Empty spaces created by deleting joints are saved into a singly connected list. These empty spaces are later used for receiving new joints during the migration of elements, see Chapter 3.11. Lastly, global IDs of new nodes are assigned, see Chapter 3.6.2.

3.7 Parallelisation of Nodal Forces

Nodal forces are calculated in each time step during time integration of equations of motion. Those forces include a force calculated from the deformation of a constant strain triangular element and also forces coming from the deformation of joint elements, through which surrounding elements act on a current element. Lastly, contacts between discrete elements introduce contact forces, see Chapter 3.8. All these forces are added together to produce the total of nodal forces.

Forces arising from the deformation of both triangular and joint elements are calculated for both internal and interfacial elements. Since interfacial elements are present on more than one processor and their forces are exchanged among processors in each time step, forces must be divided by the number of processors among which the element is shared.

The number by which the force of an interfacial element is divided can easily be calculated from the flag assigned to each element (both triangular and joint element) during domain decomposition. Since each sub-domain is confined to a rectangular

shape, it has four borders and four corners. Interfacial elements B located at right, left, top and bottom border are shared between two processors and, therefore, they are assigned flags 8, 9, 10 and 11 respectively. Interfacial elements C3 (C4) located at top-right, top-left, bottom-left and bottom-right corner are shared among three (four) processors and the flags are 12, 13, 14 and 15 (16, 17, 18 and 19) respectively. Internal elements are assigned number 4 as a flag.

The flag of the element is, as a result, not just providing information about the status of an element but also about its position. Both the division number and the location of an element can be calculated as follows:

$$\begin{aligned} idiv &= flag/4 \\ ipos &= flag\%4 \end{aligned} \tag{3.2}$$

where *idiv* is a division number and *ipos* is an element's position. Operators "/" and "%" have a C (programming language) meaning for operations on integer numbers. Both *idiv* and *ipos* are used throughout the whole parallel implementation of Y2D code, especially in contact interaction and migration of elements.

Division number *idiv* equals 1, 2, 3 and 4 for internal (A) and interfacial elements (B, C3, C4) respectively. This means it is not necessary to check the status of the element and the nodal force is simply divided by *idiv*.

3.8 Parallelisation of Contact Interaction

Contacts between triangular elements result in contact forces which are added to forces arising from the deformation of triangular and joint elements.

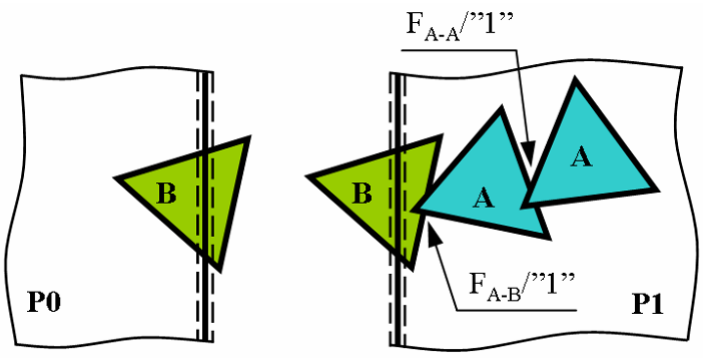


Figure 3.22: Contact force for internal elements.

Contact force between two triangular elements is divided by a number which depends on the combination of statuses of elements in contact:

- Contact between an internal element with any type of interfacial element (B, C3, C4) results in unique contact force and this force is "divided" by 1, see Figure 3.22.

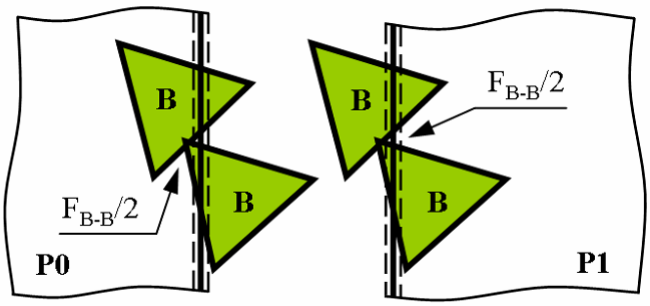


Figure 3.23: Contact force for combination of interfacial elements B-B.

- If contact between an interfacial element B and any type of interfacial element (B, C3, C4) occurs, the contact force is calculated on only two processors out of two, three or four, hence the force must be divided by 2, see Figures 3.23 and 3.24.

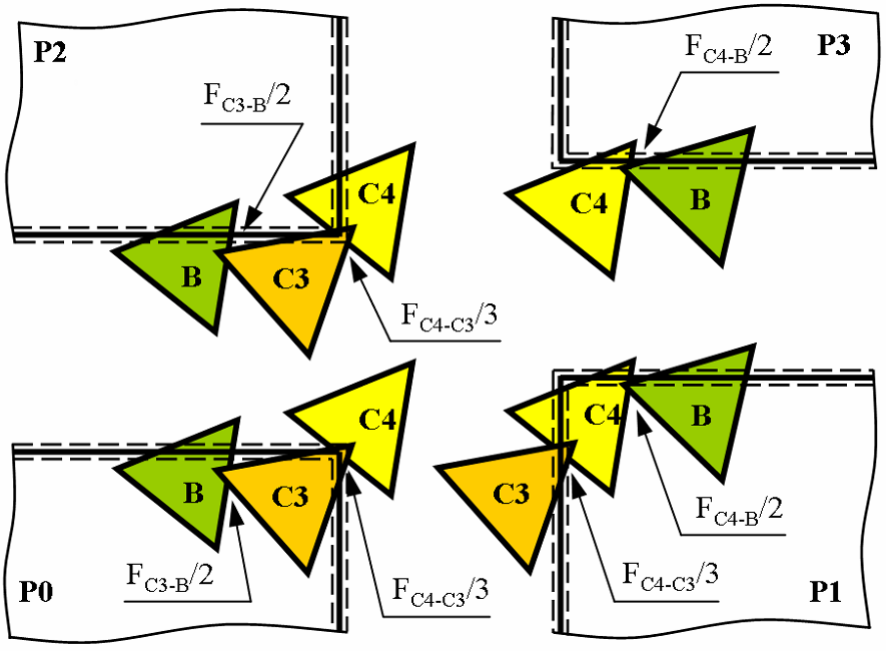


Figure 3.24: Contact forces for combinations of interfacial elements B-C3/C4 and C3-C4.

- Contact force for combination of interfacial elements C3 with C3 or C4 is calculated on three processors and the force is divided by 3, see Figure 3.24.
- Lastly contact force for combination C4-C4 must be divided by 4.

It follows from these rules that contact force between two interfacial elements must be divided by the lower division number *idiv* (Eq. 3.2) calculated from statuses of both elements, however, there are two exceptions to this rule:

- If two interfacial elements B are in contact and one element is located at the horizontal border and the second element at the vertical border, the resulting contact force must be left intact since corresponding counterparts of elements in contact are located on two different processors, see Figure 3.25a.
- In the case of contact between two interfacial elements C3, where one element is located at the corner and the second one is located at the border, the calculated contact force must be divided by 2, see Figure 3.25b.

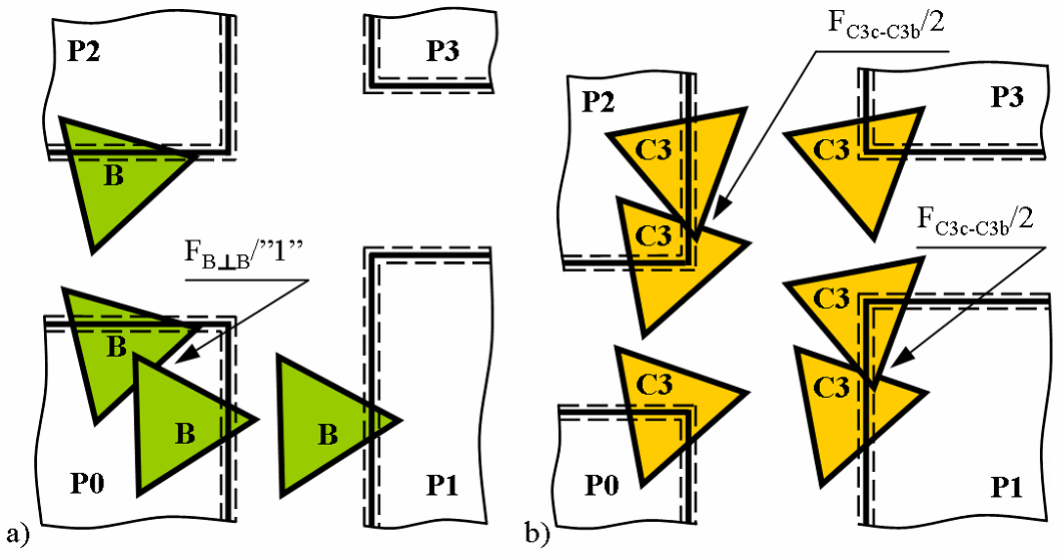


Figure 3.25: Contact forces for a) combination of interfacial elements B-B located at perpendicular borders, b) combination of interfacial elements C3-C3 located at a border and corner.

When all forces (forces calculated from deformation of triangular and joint elements and contact forces) are added together, the resulting total nodal forces of interfacial elements are exchanged between corresponding processors.

3.9 Parallelisation of Contact Detection

The rules created for parallel processing of contact interaction (see Chapter 3.8) have one significant advantage. The contact detection algorithm from a sequential FDEM code can be used directly in parallel implementation of FDEM code with only a slight modification. That means each of the processors performs the contact detection on its local sub-domain independently from other sub-domains, instead of performing contact search globally on the whole computational domain and parallelising it. As a result the parallel programming needed to implement contact detection is greatly simplified.

When the contact detection is performed in a sequential FEEM code, the singly connected lists of contacting couples are only updated in order to save CPU time, instead of assembling completely new lists. This feature can be left intact in a parallel implementation of FDEM code since the buffer around the borders of the sub-domain is derived from the contact detection buffer and migration of elements introduces only a small change in the sub-domain. The same can be applied to the re-distribution of elements during load balancing, since the partitioning algorithm is incremental and again, only a small change is introduced.

As mentioned above, only a slight modification is needed in order to parallelise the contact detection algorithm. The reason for the modification is the migration and re-distribution of elements during load balancing. These two operations do not occur in the sequential Y2D code and the sequential contact detection algorithm is therefore not equipped to deal with this situation. Two changes can be introduced by these two operations:

- Processor receives new elements located around the borders.
- Processor deletes elements that moved outside its sub-domain.

The first option does not pose any problem for the contact detection algorithm since this is analogical to a new contact between elements. Deletion of elements on the other hand is something which cannot happen in sequential Y2D code and must be dealt with. Since lists of contacting couples are only updated, the list of contacting couples for a deleted element still exists, even though the element is already deleted. The solution to this problem is quite simple: elements saved in the list of deleted elements are loaded into the contact detection grid as if they were still present in the sub-domain. Then, at the stage where the contact detection algorithm checks for elements which

are no longer in contact, an option is added to deal with the deleted elements. These elements are then simply removed from the list of contacting couples.

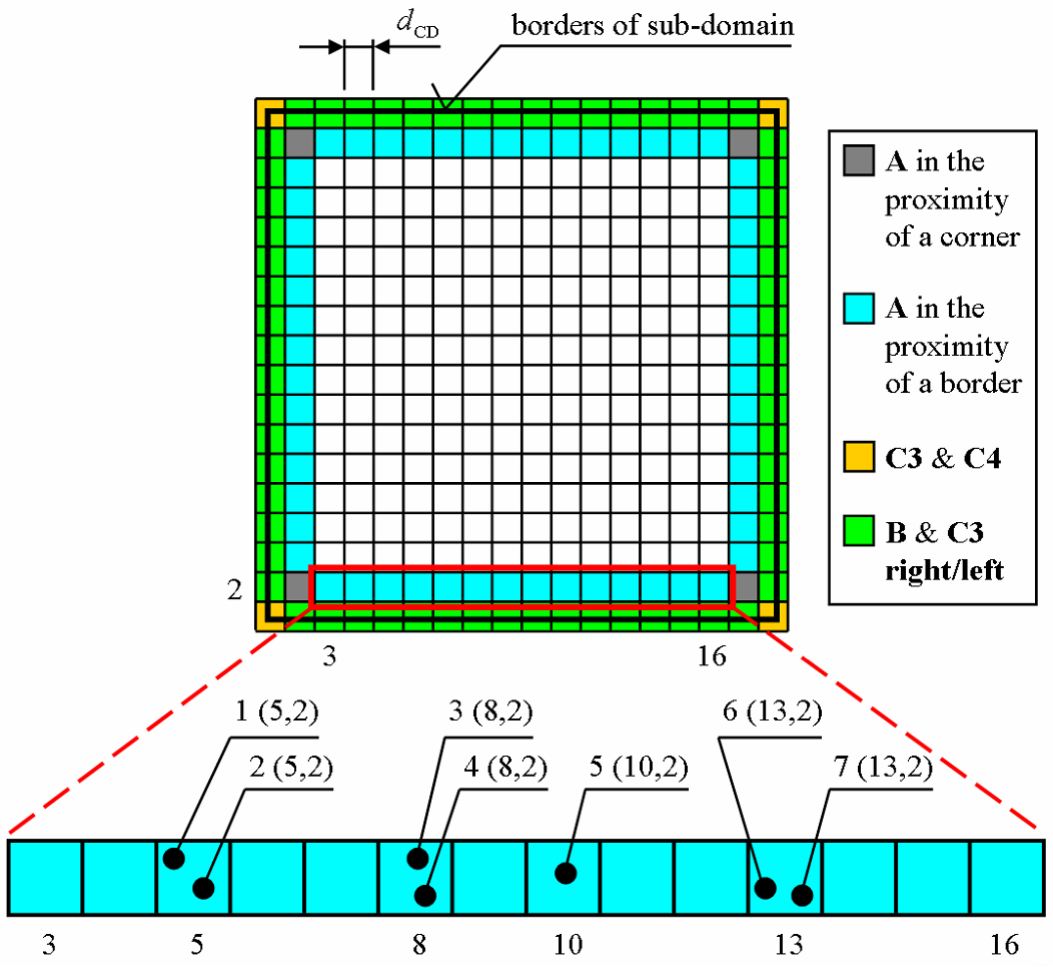


Figure 3.26: Contact detection grid with highlighted cells from which lists of internal and interfacial elements located in the proximity of each border/corner are assembled.

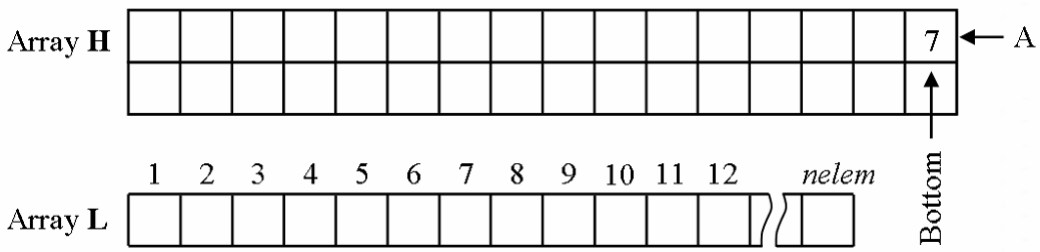


Figure 3.27: List of internal triangular elements located in the proximity of bottom border.

During contact detection the sub-domain is discretized into a contact detection grid

and a list of elements for each cell in the grid is assembled. This information is used to assemble singly connected lists of internal and interfacial triangular elements located in the proximity of each border and corner, see Figure 3.26.

Figure 3.27 shows an example of a list assembled for internal elements located in the proximity of the bottom border from a Figure 3.26. Integer array **L** contains all singly connected lists and its size is equal to the number of elements in the database *nelem*. Each cell in integer array **H** holds the first element for one singly connected list assembled during contact detection. The structure of array **H** is explained in the Figure 3.28. The size of array **H** must be calculated from the number of processors in column *icol*, since the number of segments *nsegm* at right/left border is $nsegm \leqq icol$ and the number of neighbouring borders *nbor* at right/left border (C3 at right/left border) is $nbor \leqq (icol - 1)$. In the example in Figure 3.28 *nsegm* is equal to 2 and *nbor* is equal to 1.

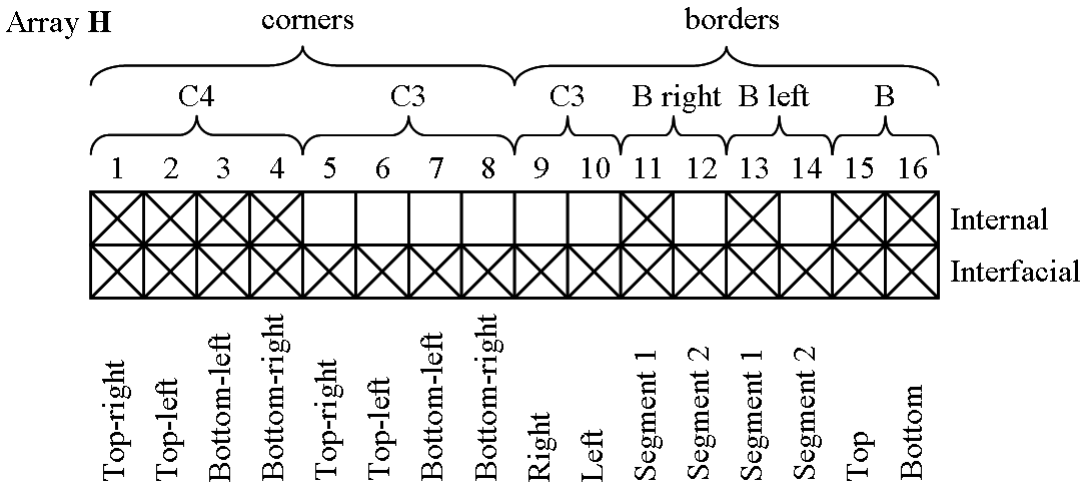


Figure 3.28: Structure of an integer array **H**.

Lists of interfacial elements for each border/corner are used to prepare messages for the exchange of total nodal forces in each time step. All lists (internal and interfacial elements) are used to update the status of elements located in the proximity of borders of the sub-domain, while performing the migration of elements.

3.10 Migration of Elements

A typical problem in the combined-finite discrete element method is a dynamic one, meaning that discrete elements are moving through the computational domain, interact

with each other through contact and fracture and fragmentation may occur.

When the domain decomposition is done at the start of the simulation, each processor is assigned a sub-domain with a fixed size. In every time step the nodal forces of interfacial elements are exchanged between corresponding processors, and equations of motion are solved resulting in new positions of elements. It is therefore necessary to take care of elements either leaving or arriving at the sub-domain. Hence, the migration of elements from one sub-domain to another is an important part of the parallel implementation of FDEM code.

Since element migration is expensive in terms of CPU time, its frequency is controlled by a buffer zone (Figure 3.3) introduced around the borders of the sub-domain (see Chapter 3.4.1) to avoid performing it in every time step. The size of the buffer zone is calculated from the buffer controlling the frequency of contact detection, see Eq. 3.1. Migration of elements is therefore performed only if the maximum travelled distance is bigger or equal to the size of the buffer zone.

The maximum distance travelled is the sum of the maximum distances calculated in each time step. After the migration of elements, this distance is set to zero. The travelled distance for all elements is calculated and the maximum value found and added to the sum in each time step. When the sum is equal to or bigger than the size of the buffer zone, migration of elements is performed and the sum is again reset to zero.

The size of the buffer zone has a conflicting impact on the performance of parallel implementation. If the buffer is small, the number of interfacial elements is also small and communication overhead in each time step is smaller, but the migration of elements must be performed with higher frequency. As previously mentioned, migration is very expensive in terms of CPU time. On the other hand, if the buffer is big, the number of interfacial elements increases, which in turn increases the communication overhead in each time step, but the migration of elements occurs less often. The minimum size of the buffer zone is equal to the buffer controlling the frequency of contact detection and the maximum size is fixed to its multiple of 4. The size of the contact detection buffer is around $1/5$ of the size of the element and therefore the maximum size calculated from the contact detection buffer is $4/5$ of the size of the element. Numerical experiments are needed to find the optimal size of the buffer zone.

3.10.1 New Position of Elements

When the migration is triggered, positions of internal triangular elements located close to borders of the sub-domain and all interfacial triangular elements must be checked,

and the status of each element, if necessary, updated. To check new positions of the elements, singly connected lists of triangular elements assembled during contact detection are used instead of checking positions of all elements in the sub-domain. The cost of migration is therefore reduced significantly since only the position of a small fraction of all elements is checked. Because lists used to check the position of elements are assembled for each border/corner separately (Figure 3.28), the approximate position of each element is known beforehand. Hence the implementation of the moving of elements is simplified.

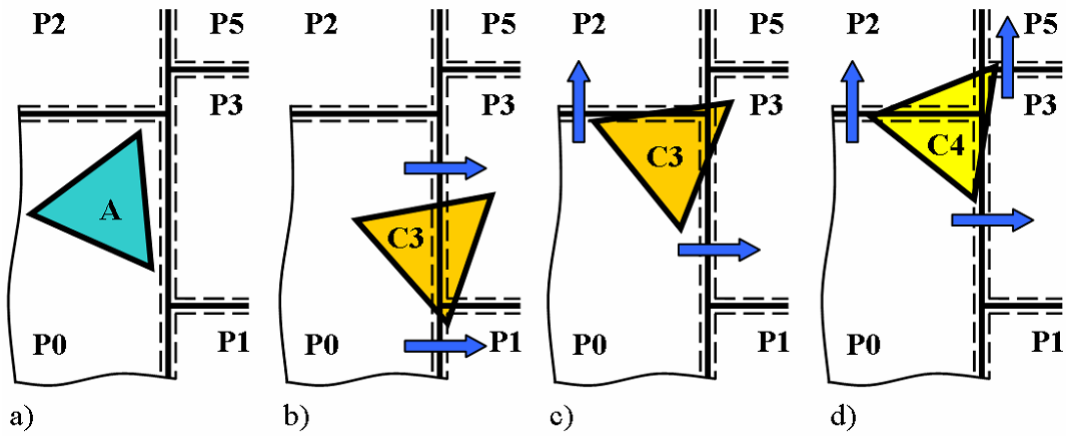


Figure 3.29: Moving of internal element A: a) original internal element A before moving, b) internal element becomes interfacial element C3 located at right border (two horizontal messages are necessary), c) A becomes C3 located at top-right corner (one horizontal and one vertical message is necessary), d) A becomes C4 located at top-right corner - element is first sent in horizontal message to processor 3 and then processors 0 and 3 send element again in vertical messages.

The following cases can occur while checking positions of internal and interfacial triangular elements:

- Internal element A can stay an internal element (no action needed) or become an interfacial element. An internal element cannot leave the sub-domain since the maximum size of the buffer zone is fixed to $4/5$ of the size of element. If it becomes an interfacial element:
  - B - element must be sent to one corresponding processor. For B located at right/left border the search must be performed to find the rank of the neighbouring processor since there can be more than one.
  - C3 located at the corner - element must be sent to two corresponding processors, see Figure 3.29c.

  - C3 located at right/left border - ranks of both receiving processors must be found, see Figure 3.29b.
  - C4 - element must be sent to three corresponding processors, see Figure 3.29d.

- Interfacial element B located at top/bottom border can stay the same, leave the sub-domain and be deleted or it can become:
  - internal element A at current processor (no message is needed), see Figure 3.31b.
  - B located at right/left border - element must be sent to the last/first segment (neighbouring processor), see Figure 3.30b.
  - C3 located at the corner - element must be sent to one processor in a horizontal message, see Figure 3.30c.
  - C3 located at right/left border - element must be sent to two processors, out of which the first one is at the last/first segment and the rank of the second one is equal to the rank of the first one $\pm icol$ depending on the grid configuration, see Figure 3.31c.
  - C4 - element must be sent to two corresponding processors in horizontal messages, see Figure 3.30d.

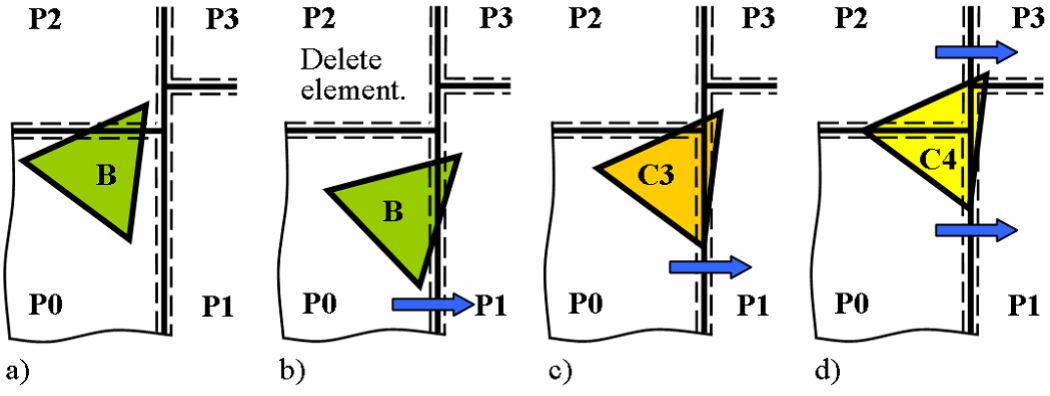


Figure 3.30: Moving of interfacial element B located at top border: a) original interfacial element B before moving, b) element becomes interfacial element B located at right border (one horizontal message is necessary), c) element becomes C3 located at top-right corner (one horizontal message is necessary), d) element becomes C4 located at top-right corner (two horizontal messages sent by processors 0 and 2 are necessary).

- Interfacial element B located at right/left border can stay the same, leave the sub-domain and be deleted or it can become:
  - internal element A at current processor (no message is needed), see Figure 3.33b.
  - B located at top/bottom border - element must be sent to the neighbouring top/bottom processor, see Figure 3.32b.
  - C3 located at the corner - element must be sent to one neighbouring processor in a vertical message, see Figure 3.32c.
  - C3 located at right/left border - no message is needed on current processor, see Figure 3.33c.
  - C4 - element must be sent to two corresponding processors in vertical messages, see Figure 3.32d.
- Interfacial element C3 located at any corner can stay the same or it can become internal element A, interfacial element B located at any border or leave the sub-domain in which case the element must be deleted. For all those cases no message is needed and status of element is just updated on current processor. C3 can also become interfacial element C4 or C3 located at right/left border. In both cases one message in vertical direction, sent from the neighbouring processor, is needed, see Figure 3.34.

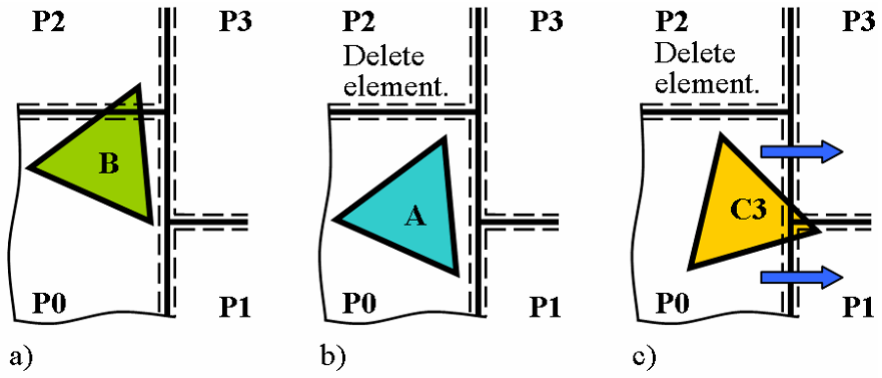


Figure 3.31: Moving of interfacial element B located at top border: a) original interfacial element B before moving, b) element becomes internal element A (element is deleted on processor 2), c) element becomes C3 located at right border (two horizontal messages are necessary and element is deleted on processor 2).

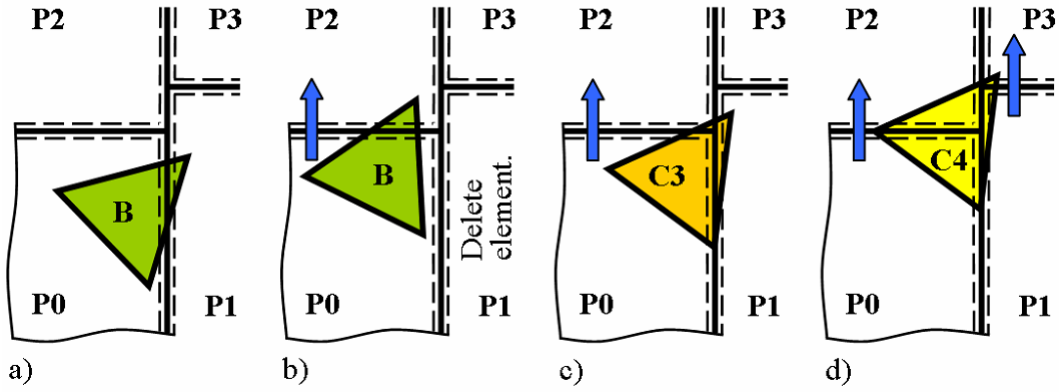


Figure 3.32: Moving of interfacial element B located at right border: a) original interfacial element B before moving, b) element becomes interfacial element B located at top border (one vertical message is necessary), c) element becomes C3 located at top-right corner (one vertical is message necessary), d) element becomes C4 located at top-right corner (two vertical messages sent by processors 0 and 1 are necessary).

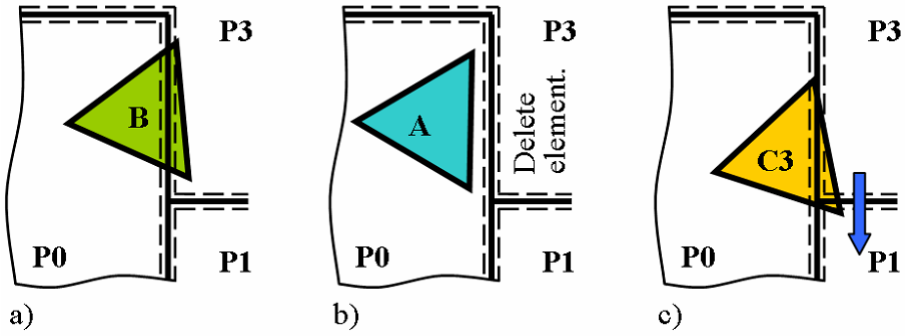


Figure 3.33: Moving of interfacial element B located at right border: a) original interfacial element B before moving, b) element becomes internal element A (element is deleted on processor 3), c) element becomes C3 located at right border on processor 0 (one vertical message sent from processor 3 to processor 1 is necessary).

- Interfacial element C4 located at any corner can stay C4 or it can become any other element (internal A or interfacial B, C3) or it can leave the sub-domain and necessarily be deleted. No message is needed for any change of status.

The position of a triangular element is checked by using x or y coordinates of all three nodes against the vertical or horizontal border respectively, or both when the element is located in the proximity of a corner. If the element's new status requires sending a message, it is saved in a singly connected list of elements which must be communicated to neighbouring processors. Similarly for the lists in Figure 3.28, these lists are assembled separately for internal and interfacial elements and for each border

and corner, see Figure 3.35. The size of arrays is analogical to those assembled during contact detection.

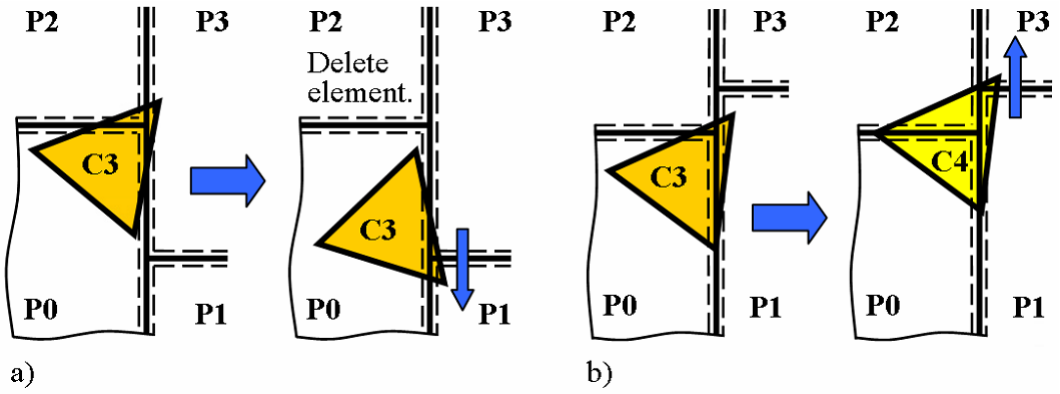


Figure 3.34: Moving of interfacial element C3 located at top-right corner: a) element becomes interfacial element C3 located at right border (element is deleted on processor 2 and sent from processor 3 to processor 1), b) element becomes C4 located at top-right corner (one vertical message sent from processor 1 to processor 3 is necessary).

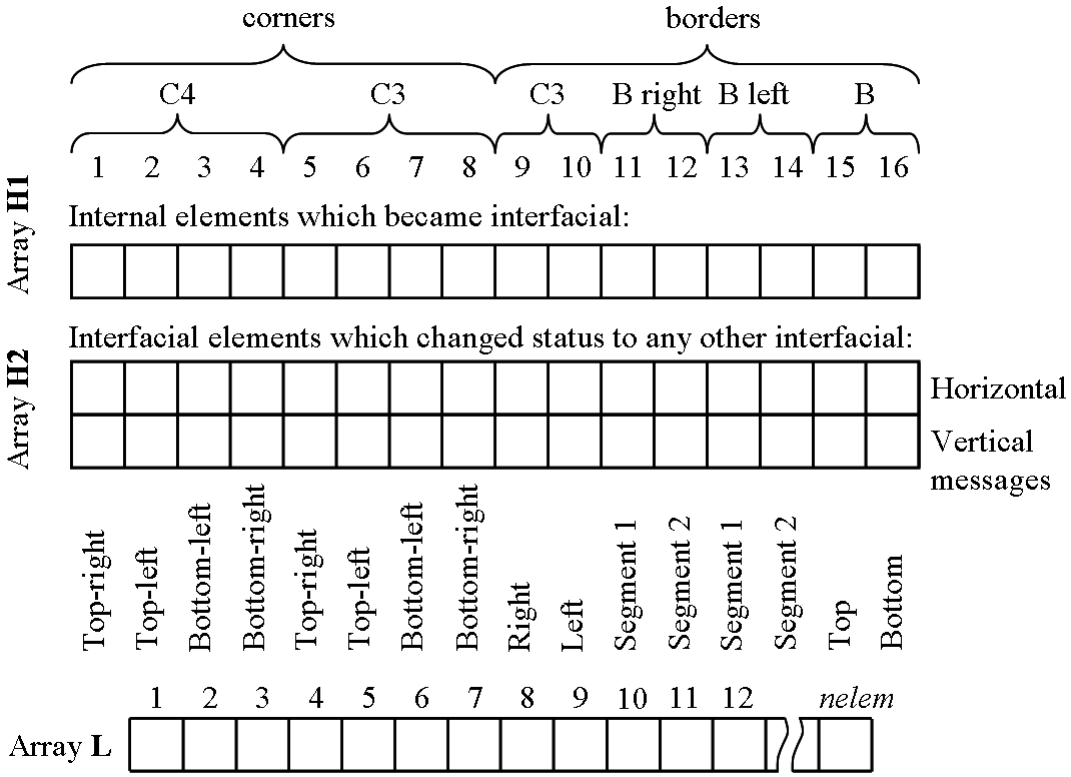


Figure 3.35: Structure of integer arrays holding first element in each singly connected list for internal (integer array **H1**) and interfacial (integer array **H2**) elements. The rest of each list is saved in an integer array **L**.

If an element leaves the sub-domain and must be deleted from the element database, it is first saved in the list as it may be required to send it in the message before deleting. Therefore, all elements are deleted after all communication is done by using the list assembled, while new positions of elements were checked.

Floating point numbers. As mentioned in Chapter 2.7 real numbers are represented in a computer's memory as floating point numbers. Due to the limited amount of memory most real numbers are therefore represented only by an approximate value. This introduces the so called "rounding error".

Due to the rounding error one can obtain different results for simple arithmetic operations on real numbers, like adding or multiplying just by changing the order in which the real numbers are processed. This is the reason results differ if exactly the same FDEM simulation is run on a different number of processors.

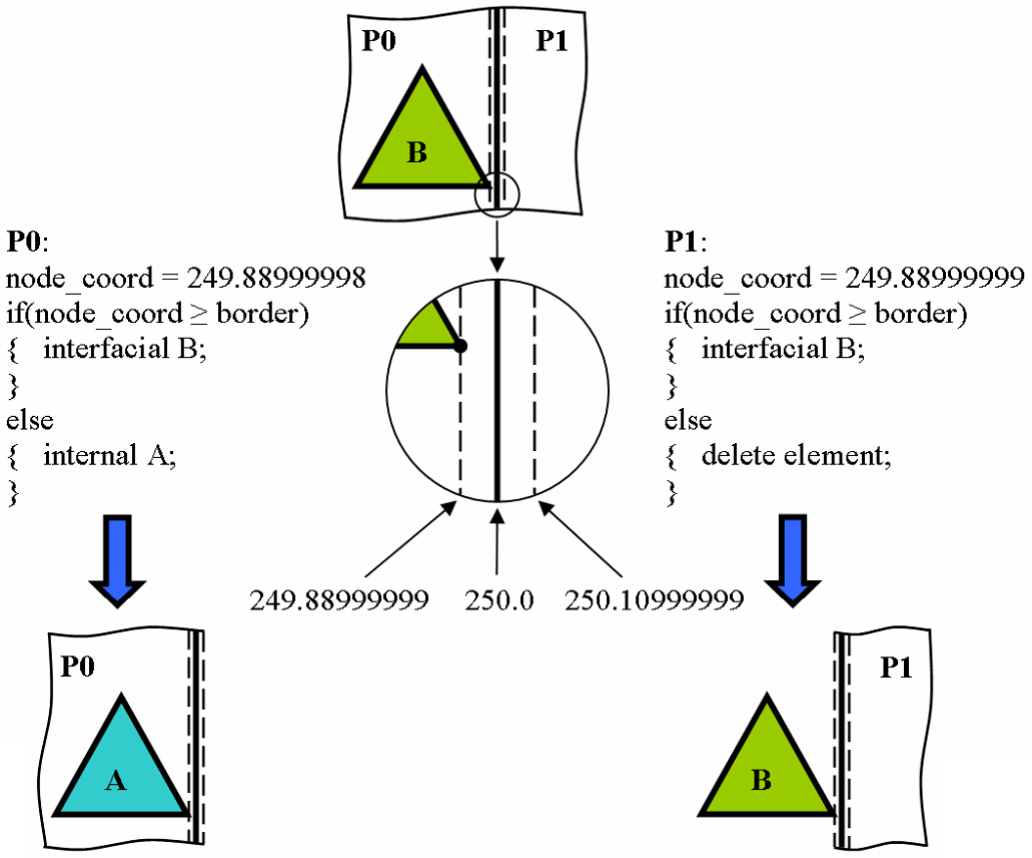


Figure 3.36: Status of element decided differently on two neighbouring processors due to the rounding error.

Another problem arises when the coordinate of an element's node must be checked against a border among two neighbouring processors. If both numbers are nearly the

same (difference between them is at the order of machine epsilon ε_M) and both processors have to process an example from Figure 3.36, then each processor can evaluate this situation with a different result due to the rounding error. In each time step this rounding error arises during exchange of total nodal forces of interfacial elements, since the forces are added on a different processors in different order. This could lead to a situation when, on the first processor, an element is assigned status as internal element A while on the other processor status would be an interfacial element B, see Figure 3.36. Different status of the same element on two neighbouring processors would cause the simulation to fail. These kinds of problems are avoided if one master processor is making decisions and distributing results to the remaining processors. This is not the case of this parallel implementation of Y2D code.

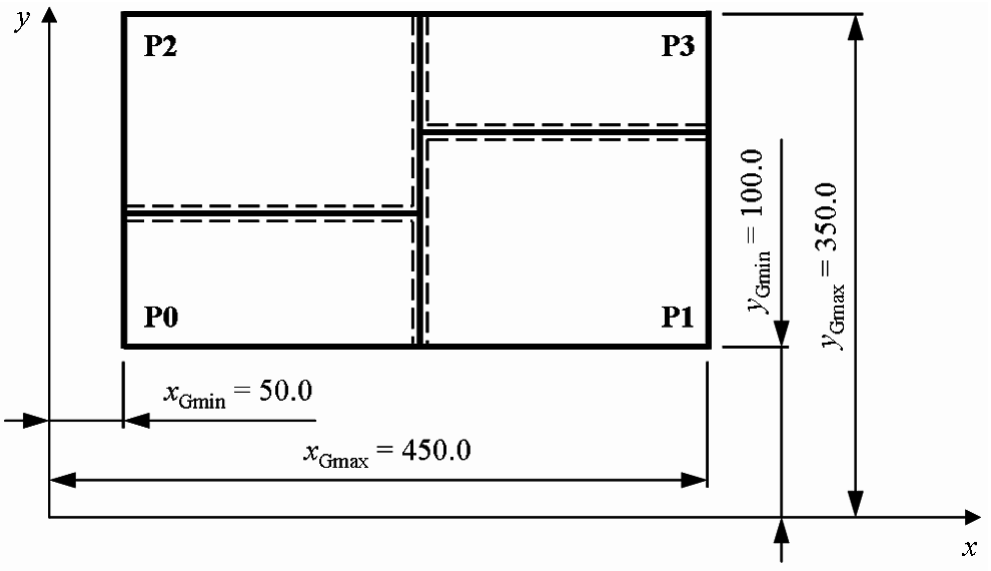


Figure 3.37: Global minimum and maximum x and y values of the computational domain.

To resolve the problem described above, all coordinates are multiplied by a big integer number and the result is integerized. This is done for all borders and all nodes belonging to triangular elements located in the proximity of borders of sub-domain (saved in singly connected lists, see Figure 3.28). The integer number by which coordinates are multiplied cannot be chosen randomly but must first be determined to make sure that the maximum integerized number is smaller or equal to the limit value for a particular data type, in this case *long int*. For example positive limit value LONG_MAX for type *long int* equals 2147483647 (or greater)[3] in 32-bit environment. Then the integer number I_{mult} for positive values can be calculated as follows:

$$I_{mult1} = (long\ int)\,(\text{LONG\_MAX}/x_{Gmax})$$
$$I_{mult2} = (long\ int)\,(\text{LONG\_MAX}/y_{Gmax}) \tag{3.3}$$

where $x_{Gmax}$and y_{Gmax} are positive maximum global x and y coordinates and $(long\ int)$ is a conversion from a floating point number to an integer number. After I_{mult1} and I_{mult2} are calculated I_{mult} is taken as minimum value from both. The same calculation is repeated for negative values by using LONG_MIN which equals -2147483647 (or less)[3] in 32-bit environment. The value of I_{mult} is therefore system dependent.

If we take the maximum global coordinate from Figure 3.37 ($x_{Gmax} = 450.0$) then I_{mult} equals 4772185. By using I_{mult} in the example from Figure 3.36 both integerized border coordinate and integerized node coordinate on processors 1 and 2 are calculated and all equal 1192521309, hence the resulting status on both processors is the same, see Figure 3.38.

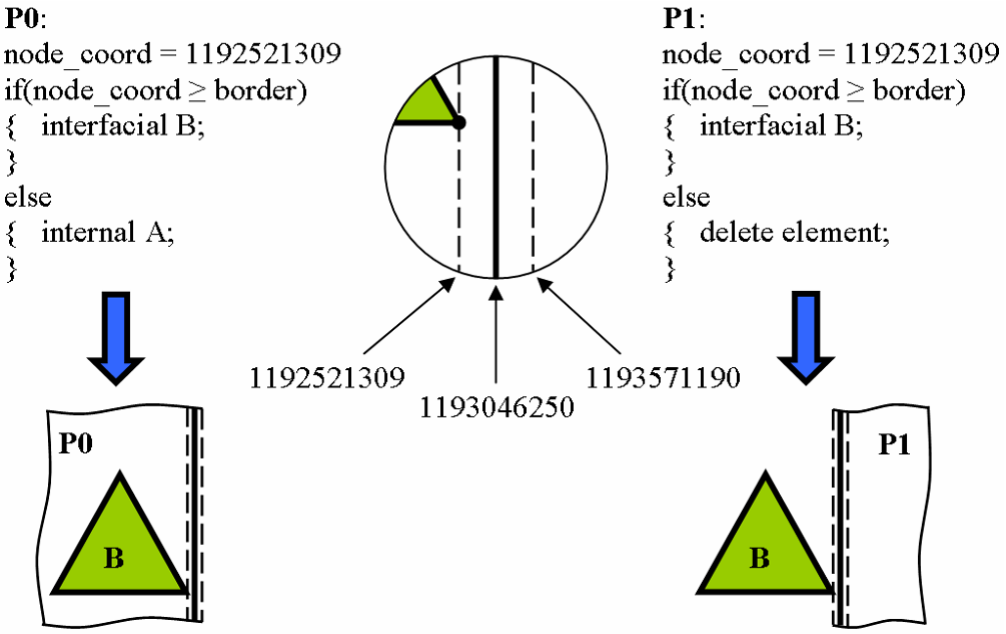


Figure 3.38: All coordinates multiplied by I_{mult} and integerized resulting in elimination of rounding error. Status of element is the same on both processors.

Joint elements. When the positions of all triangular elements located in the proximity of borders of the sub-domain have been checked, and if necessary, a new status assigned to each, the status of joint elements is also updated according to rules outlined in Chapter 3.4.3. For this lists of joint elements located close to each border/corner are assembled by using lists of triangular elements (Figure 3.28) in combination with

pointers from each joint element to its two triangular elements. Therefore, the status of only a small fraction of all joint elements is checked analogically to triangular elements. Also, the status changes of joint elements are analogical to triangular elements, as well as lists of joints which must be communicated to neighbouring processors.

3.10.2 Communication

After the statuses of triangular and joint elements located in the proximity of borders of the sub-domain have been updated, elements which changed status from either internal A to interfacial or from interfacial (B, C3) to interfacial with a higher number of processors (C3, C4) must be sent to neighbouring processors.

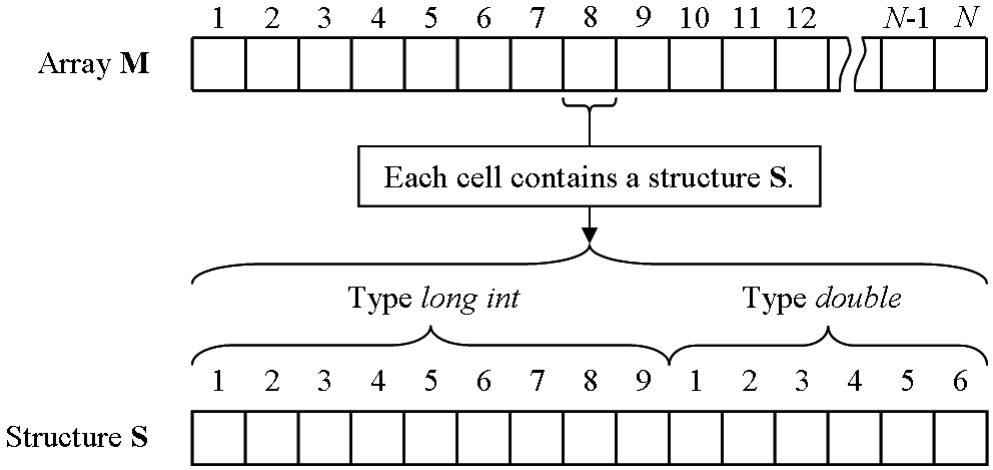


Figure 3.39: Data structure of a message: message is saved in an array **M** where each cell contains a structure **S**.

To successfully exchange and receive messages it is necessary to prepare:

- Derived datatype *tyus* - this is a MPI construct to accommodate for messages with a more complicated structure. To send an element with its nodes requires copying a mixture of integer and floating point numbers into a message. This kind of message can be sent only if a derived datatype is built and this datatype is then passed as an argument to the communication function. The data for the message are saved in an array **M** of size N where each cell contains a structure **S** of 9 integer numbers (type *long int*) and 6 floating point numbers (type *double*), see Figure 3.39. The size of the structure **S** is enough to hold all data associated with sending one element (both triangular and joint element) or one node. For this structure the derived datatype *tyus* is built (see Chapter 2.4.2 for general

information about derived datatypes) by using information from the Figure 3.40, i.e. number of blocks, block length, type and displacement of each block. One of the arguments in the communication function is the size of the array *N* where for each cell the derived datatype *tyus* is passed as a type. In order to save CPU time the derived datatype *tyus* is *static* (C++ meaning) and it is built only once when the first moving of elements is done.

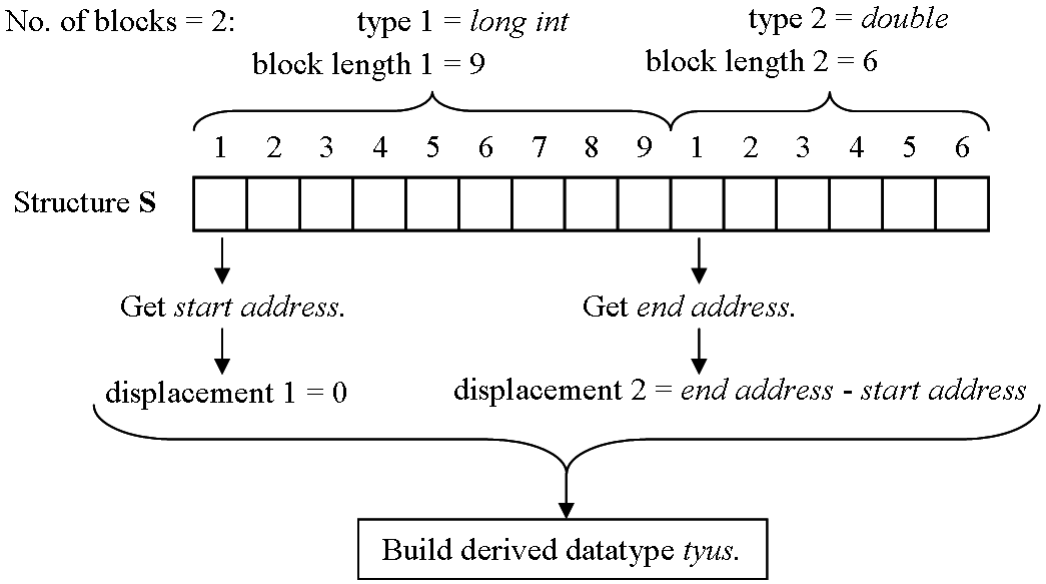


Figure 3.40: Information used to build a derived datatype *tyus* from a structure **S**.

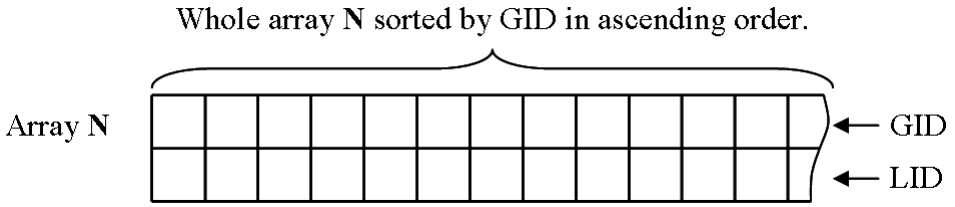


Figure 3.41: Global and local IDs of nodes belonging to elements located close to borders of sub-domain saved in an integer array **N**.

- List of nodes which belong to elements located in the proximity of borders of sub-domain. This list is made by using singly connected lists assembled during contact detection (see Figure 3.28). List of nodes is saved in an integer array **N** (see Figure 3.41) where global IDs (GID) of nodes are saved in the first row and local IDs (LID) of nodes are saved in the second one. Global IDs are then sorted in ascending order and values in the second row are rearranged together

with values in the first row. The array **N** is later used to receive new nodes and elements.

The communication is first performed in the horizontal direction and in the second stage in the vertical direction. For further details on a communication engine see Chapter 4.12. The messages for each communication are prepared by using singly connected lists (see Figure 3.35) which were assembled while checking new positions of internal and interfacial elements. Elements in these lists must be communicated in a way indicated by blue arrows in Figures 3.29-3.34.

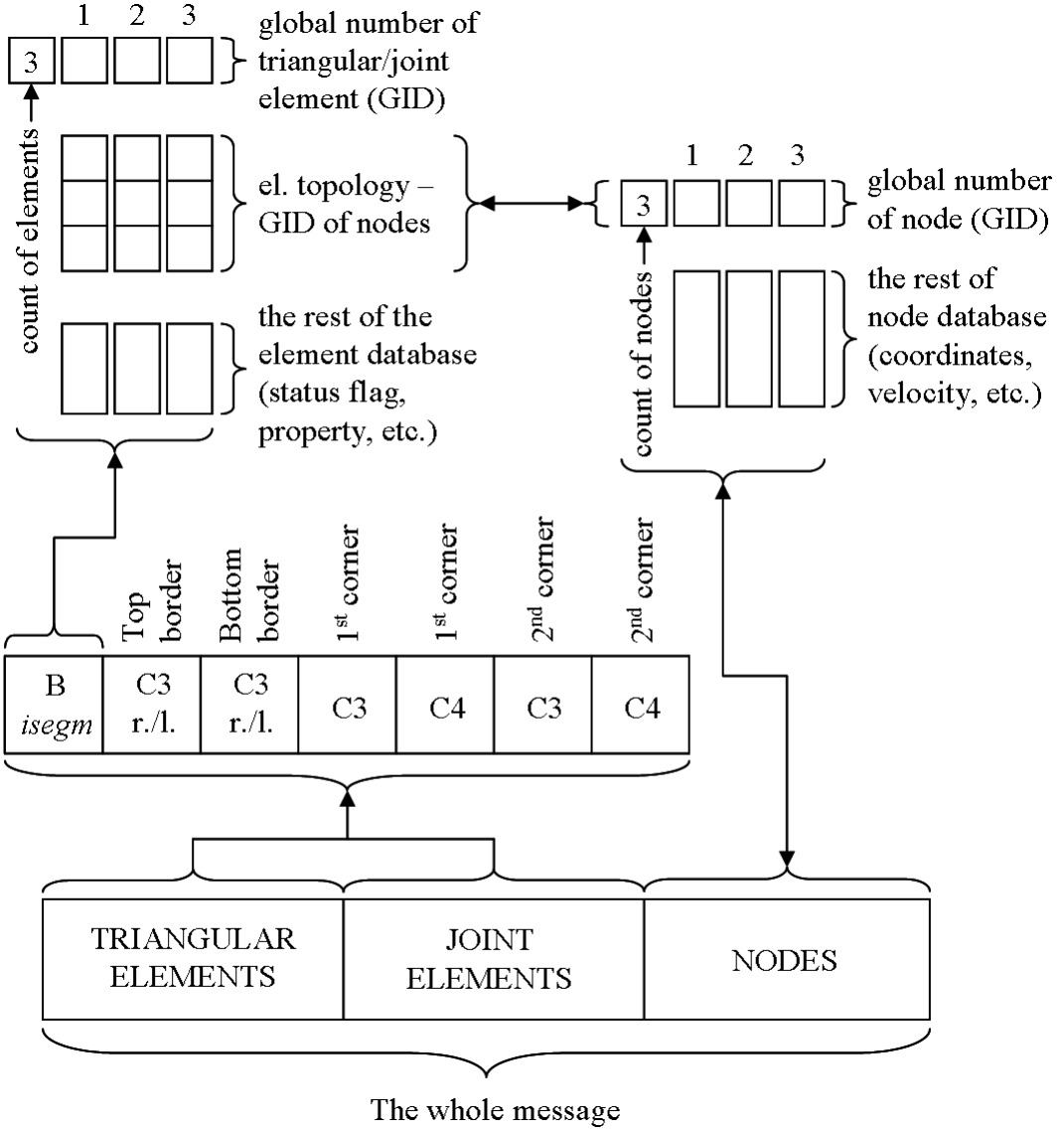


Figure 3.42: Structure of the whole message containing triangular and joint elements and their nodes.

Elements and nodes for each message are saved in the array **M** (see Figure 3.39) and are organised into three main blocks: triangular elements, joint elements and nodes, see Figure 3.42. Triangular and joint element blocks are further divided into seven sub-blocks containing elements with status: interfacial elements B for a segment *isegm*, C3 right/left located at top border of *isegm* (see Figure 3.43), C3 right/left at bottom border, C3 and C4 at first corner and C3 and C4 at second corner. Elements belonging to each sub-block are therefore saved to the message in the order mentioned above (Figure 3.42).

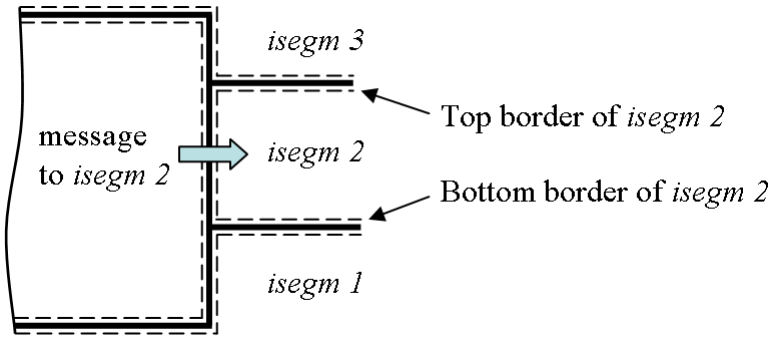


Figure 3.43: Preparing a message: top and bottom border of a segment (neighbouring processor).

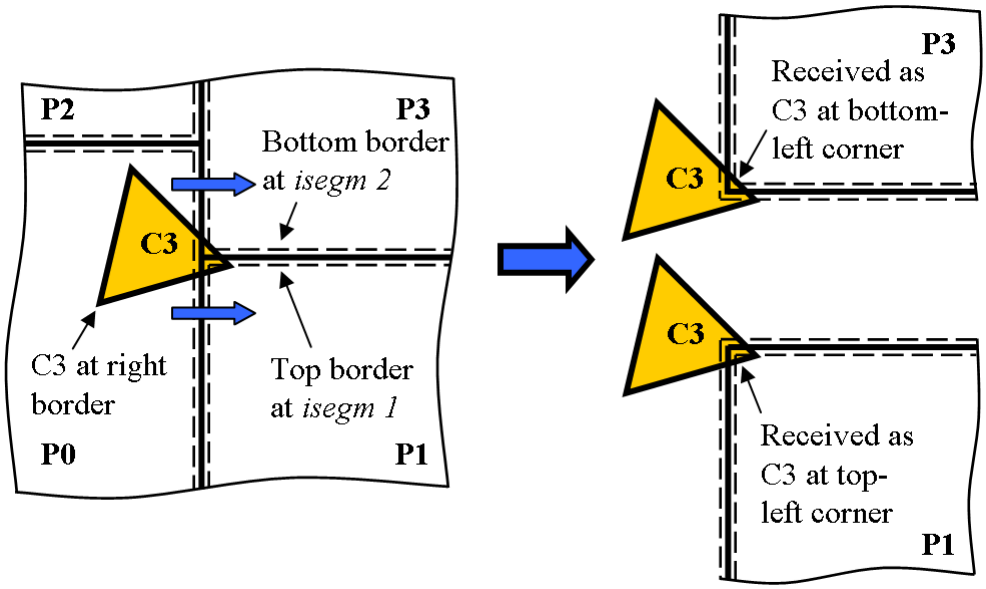


Figure 3.44: Status of interfacial element C3 at sending and receiving processors.

The reason for structuring the message in the way described above is that the status of each element in each block is known on the receiving processor straight away. For

instance, if the interfacial element C3 located at the right border and top border of *isegm 1* (first message) and the bottom border of *isegm 2* (second message) is sent to the remaining two neighbouring processors, the status on the first receiving processor will be C3, located at the top-left corner, and the status on the second one will be C3, located at the bottom-left corner, see Figure 3.44. The remaining statuses are resolved on receiving processors analogically depending on the position of the element in the sub-domain, see Figures 3.29-3.34.

When an element (triangular or joint element) is copied into the message, only its global ID is saved. Local IDs in its element topology (numbers of nodes which belong to the element) are replaced by global IDs and nodes are saved into the message with their global IDs as well, see Figure 3.42. Pointers from joint elements to their triangular elements (local IDs of triangular elements) are also copied into the message with global IDs.

After horizontal/vertical messages are exchanged between processors, content of the message must be saved into the element and node databases, see Algorithm 3.3. Since only new elements (triangular and joint elements) are received, there is no need to check if some element is already present in the database and element is directly saved into element database. New triangular elements are first saved into empty spaces created in previous time steps (if an element leaves the sub-domain, it is deleted and the empty space is saved into a singly connected list) and if empty spaces are filled the remaining new elements are saved at the end of the database. The element is saved to the position at $nelem$ ($nelem$ is the number of triangular elements saved in the local database on each processor) and $nelem$ is increased by one. Joint elements are handled analogically. Received elements (both triangular and joint elements) are saved into lists (integer arrays **TEL** and **JEL**, see Algorithm 3.3) and after new nodes are received, these lists are used to replace global IDs of nodes in element topology with local IDs (actual position of a node in the node database). Global IDs are searched for in an integer array **N2** (see paragraph below) by using a binary search and, when found, replaced by local IDs.

Unlike elements it is possible for new nodes received in the message to be already present in the node database. This is because nodes are shared by more than one element and one of those elements (and its nodes) can be located within the sub-domain. Therefore, it is necessary to search for each new node in the integer array **N** (Figure 3.41) and then, only if the node is not found, it is saved in the node database.

Algorithm 3.3 Saving content of a received message into the database.

```
1:  integer **N2;                          ▷ 2D array containing GID and LID of new nodes.
2:  integer *TEL;                          ▷ List of received new triangular elements.
3:  integer *JEL;                          ▷ List of received new joint elements.
4:  integer **PJ;                          ▷ List for replacing pointers of new joints.
5:  while reading message do
6:      while new nodes do ▷ Copying nodes from the message to the node database.
7:          Search for GID of node in array N;                    ▷ See Algorithm 3.4.
8:          if GID not found then          ▷ Node is not present in the node database.
9:              if empty space available then
10:                 Save node to empty space;
11:             else                                   ▷ All empty spaces filled.
12:                 Save node at the end of node database;
13:                 nnopo = nnopo + 1;
14:             end if
15:             Save GID and LID of node to array N2;
16:         end if
17:     end while
18:     while new elements do ▷ Copying triangular/joint elements from the message.
19:         if empty space available then
20:             Save element to empty space;
21:         else                                       ▷ All empty spaces filled.
22:             Save element at the end of element database;
23:             nelem = nelem + 1;
24:         end if
25:         Assign status of element;   ▷ Status for each block of elements is known.
26:         if triangular element then
27:             Save element's LID to array TEL;
28:             Save element's LID and GID to array PJ;
29:         else                                              ▷ Joint element.
30:             Save element's LID to array JEL;
31:         end if
32:     end while
33: end while
34: Sort arrays N2 and PJ by GID in ascending order;
35: while elements in TEL and JEL do              ▷ Loop over received new elements.
36:     while nodes in element's topology do
37:         Search GID of node in array N2 and when found replace by LID;
38:     end while
39:     if joint element then
40:         Search GID of its pointers in array PJ and when found replace by LID;
41:     end if
42: end while
```

Analogically to elements, new nodes are also first saved in empty spaces (created in previous time steps by deleting nodes) and if those are filled new nodes are saved at the end of the node database. Global and local IDs of new nodes are saved in an integer array **N2** (analogical to the array **N** in Figure 3.41) and the array is sorted by global IDs in ascending order. This array is used to replace global IDs in element topology (see paragraph above)

As mentioned above, joint elements are received analogically to triangular elements. Global IDs in element topology are also replaced in the same way, but there is one more issue with receiving new joints. Each joint element is sent with global IDs of its triangular elements and again these must be replaced with local IDs (position in the database). For this reason, the global and local ID of each new triangular element is saved in an integer array **PJ** analogically to the array **N** (Figure 3.41) and sorted by global IDs in ascending order. After new joints are received, global IDs in their topology are searched for and replaced by local IDs of triangular elements.

The whole process of saving the content of a message into the computer's memory is summarised in Algorithm 3.3 and the whole communication process is summarised in the Algorithm 3.4.

Algorithm 3.4 Communication process.

1: **integer** $**$**N**; ▷ 2D array containing GID and LID of nodes around borders.
2: **if** *moving first time* **then** ▷ If moving elements first time.
3: *Build derived datatype*; ▷ See Figure 3.40.
4: **end if**
5: *Prepare array* **N**; ▷ LID, GID of nodes around borders, see Figure 3.41.
6: *Prepare horizontal messages*; ▷ See Figure 3.42.
7: *Send and receive horizontal messages*; ▷ Function MPI_Sendrecv().
8: *Read new elements and nodes from the message*; ▷ See Algorithm 3.3.
9: *Replace GID with LID* (*element topology*, *joint's pointers*); ▷ See Algorithm 3.3.
10: *Prepare vertical messages*; ▷ See Figure 3.42.
11: *Send and receive vertical messages*; ▷ Function MPI_Sendrecv().
12: *Read new elements and nodes from the message*; ▷ See Algorithm 3.3.
13: *Replace GID with LID* (*element topology*, *joint's pointers*); ▷ See Algorithm 3.3.

3.10.3 Migration of Broken Joint Elements

As mentioned in Chapter 3.4.3, a joint element represents a discrete crack model in order to enable fracture in FDEM and acts as a bond between two triangular elements. If fracture occurs, the bond is severed and the joint element is broken.

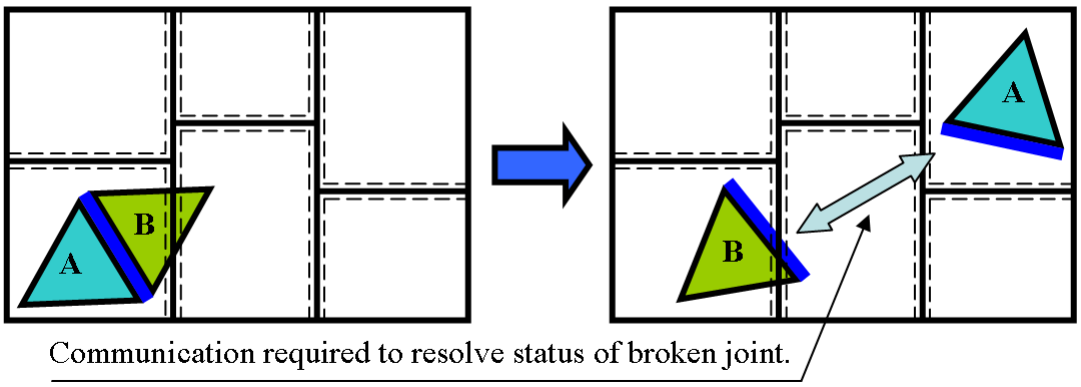


Figure 3.45: If broken joint is not deleted, determining the status of broken joint would be very costly in terms of CPU time.

Algorithm 3.5 Broken joints.

```
1:  Calculation of nodal forces :
2:  if joint breaks then                          ▷ Opening and sliding displacements too big.
3:      Get statuses of both triangular elements;
4:      if both triangular elements internal then
5:          Set flags for both edges to 1;
6:          Delete broken joint from database;
7:      else                                      ▷ At least one triangular element is interfacial.
8:          Set property of joint < 0;            ▷ To distinguish from remaining joints.
9:          Split broken joint into 2 joints;     ▷ One joint on each edge.
10:         Save both joints to list of broken joints;   ▷ Used for deleting later.
11:     end if
12: end if
13: ⋮
14: Moving of elements :
15: if joint on edge then                         ▷ Joint has only 1 triangular element.
16:     Prepare for message;      ▷ Put into list and then copy the flag into message.
17: end if
18: Send and receive messages;                    ▷ Horizontal and vertical.
19: Save flag received in the message;            ▷ Receiving processor.
20: Delete broken joints and set flags;           ▷ Sending processor.
```

Sequential FDEM code deals with broken joints by simply setting its property to a number smaller than zero, which means the joint element stays in the element database. This would not work in a parallel version of FDEM code since the bond between triangular elements is broken and, during the run of the simulation, both triangular elements could move to sub-domains which are far from each other (sub-domains are not neighbours), see Figure 3.45. It would, therefore, be extremely difficult to maintain

the status of the joint which is given by the combination of statuses of its triangular elements. This would require some additional communication resulting in the increase of the cost of parallelisation in terms of CPU time. This is not justifiable since the joint has no influence on the results of the simulation once it is broken.

When the joint gets broken, it is deleted from the element database and two flags are saved for two edges which were previously connected by the joint, see Figure 3.46. The function of the flag is to enable plotting of cracks during post-processing.

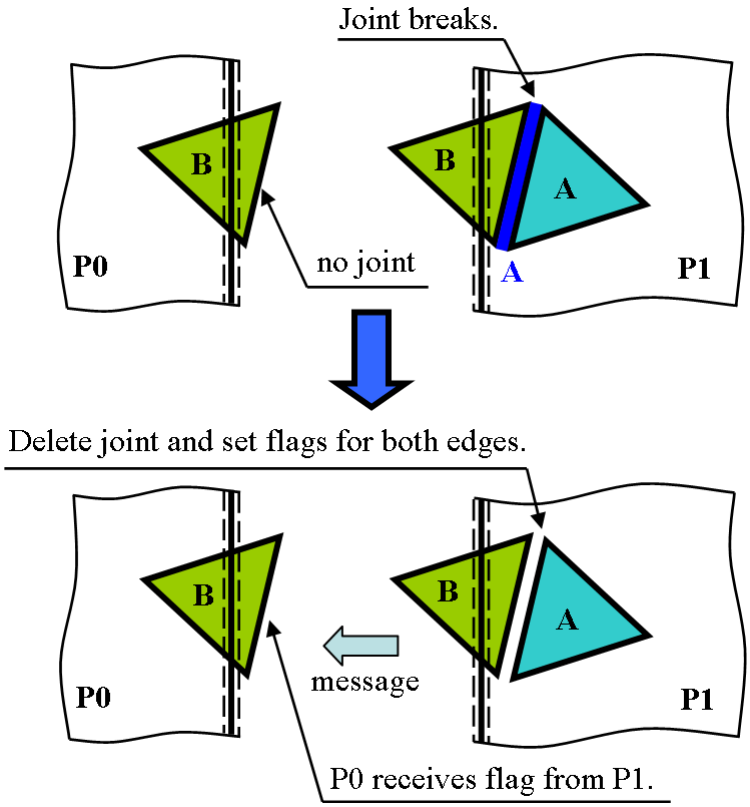


Figure 3.46: Communication is required if at least one triangular element of a broken joint is interfacial.

Fracturing occurs during the calculation of nodal forces. Opening and sliding displacements are calculated and, if they are too big, the joint is set as broken. If the joint belongs to two internal triangular elements it is deleted and the flag for each edge is set. If ,however, the joint belongs to at least one interfacial element (B, C3, C4) it cannot be deleted straight away. Instead the joint is replaced by two temporary joints placed at each edge. For instance, if the nodes on the first edge are 0 and 1, then the topology of the temporary joint will be 0-1-1-0 and the joint is therefore not acting as a bond between two triangular elements. Information that the joint is broken must be

sent to the neighbouring processor and after the communication both temporary joints can be deleted and flags on edges set. This is done when the next moving of elements occurs.

Algorithm 3.6 Moving of elements.

```
1:  integer **H1;                          ▷ 1st elements (internal) in lists (Figure 3.35).
2:  integer **H2;                          ▷ 1st elements (interfacial) in lists (Figure 3.35).
3:  integer **L;                           ▷ Remaining elements in lists (Figure 3.35).
4:  integer **HJ1;                         ▷ 1st joints (internal) in lists (Figure 3.35).
5:  integer **HJ2;                         ▷ 1st joints (interfacial) in lists (Figure 3.35).
6:  integer **LJ;                          ▷ Remaining joints in lists (Figure 3.35).
7:  integer **PTtoJ;                       ▷ Pointers from triangular to joint elements.
8:  if moving elements first time then
9:      Find I_mult;                       ▷ See Equation 3.3.
10: end if
11: Integerize coordinates of nodes;       ▷ Nodes of elements from Figure 3.28.
12: while new positions of triangular elements; do   ▷ By using array H from Figure
    3.28.
13:     if element changes status then
14:         if old status == internal then
15:             Save into H1;              ▷ Use later to check status of joints and prepare
    messages.
16:         else                           ▷ Interfacial element.
17:             Save into H2;              ▷ Use later to check status of joints and prepare
    messages.
18:         end if
19:     end if
20: end while
21: Prepare array PTtoJ;                   ▷ By using array H from Figure 3.28.
22: while triangular elements with new status do   ▷ By using arrays H1, H2.
23:     Check status of joint elements;    ▷ Use pointers in array PTtoJ.
24:     if element changes status then
25:         if old status == internal then
26:             Save into HJ1;             ▷ Including broken joints, see Algorithm 3.5.
27:         else                           ▷ Interfacial joint element.
28:             Save into HJ2;             ▷ Including broken joints, see Algorithm 3.5.
29:         end if
30:     end if
31: end while
32: Communication;     ▷ See Algorithms 3.3 and 3.4 for details. Use arrays H1, H2,
    HJ1, HJ2, L and LJ to assemble messages.
33: Delete elements and nodes;  ▷ Elements which left sub-domain (including broken
    joints, see Algorithm 3.5).
```

The whole process of handling broken joints is summarised in the form of a pseudo-code in Algorithm 3.5.

Summary. An Algorithm 3.6 summarises the whole procedure of moving of elements in the form of a pseudo-code.

Since elements which left the sub-domain are deleted and new elements which moved to the sub-domain from neighbouring processors are received during the process of moving of elements, new contact detection must, therefore, be performed. Singly connected lists of contacting couples are not re-built from scratch but rather only updated to reflect changes in the sub-domain. Singly connected list of elements located in the proximity of borders of sub-domain which are saved in an integer array **H** (see Figure 3.28) are also updated.

3.11 Load Balancing

When the first domain decomposition is performed at the start of the simulation, each sub-domain is assigned to a single processor in the PC cluster. The position of each element is changing in an unpredictable way during the run of the simulation and elements can move from one sub-domain (processor) to another assuming the size of each sub-domain does not change.

Migration of elements among processors creates an imbalance in the workload of processors. If the imbalance exceeds a maximum value specified in the input file the re-partitioning (size of each sub-domain is updated by using modified RCB algorithm,[174] see Chapter 3.5) and load balancing is carried out. The whole process is described in the following steps:

Step 1. In order to perform re-partitioning of the computational domain, the load balancing (LB) grid must be assembled. This is done during contact detection if the calculated workload imbalance exceeds the specified maximum value. Workload on i-th processor W_i is given by:

$$W_i = nelem + N_{CC} \tag{3.4}$$

where $nelem$ is a number of elements saved in the element database on i-th processor and N_{CC} is a number of contacting couples on i-th processor.

The workload on each processor is gathered by using the MPI_Allgather() function and then the imbalance B is calculated as follows:[192]

$$B = \frac{W_{max} - W_{min}}{\overline{W}} \tag{3.5}$$

where W_{max} is a maximum workload, W_{min} is a minimum workload and $\overline{W}$ is an average workload given by:

$$\overline{W} = \frac{\sum_{i=1}^{n} W_i}{n} \tag{3.6}$$

where W_i is a workload on i-th processor and n is a number of processors.

Load balancing is then triggered if the workload imbalance B exceeds the maximum imbalance B_{max} specified in the input file

$$B \geqq B_{max} \tag{3.7}$$

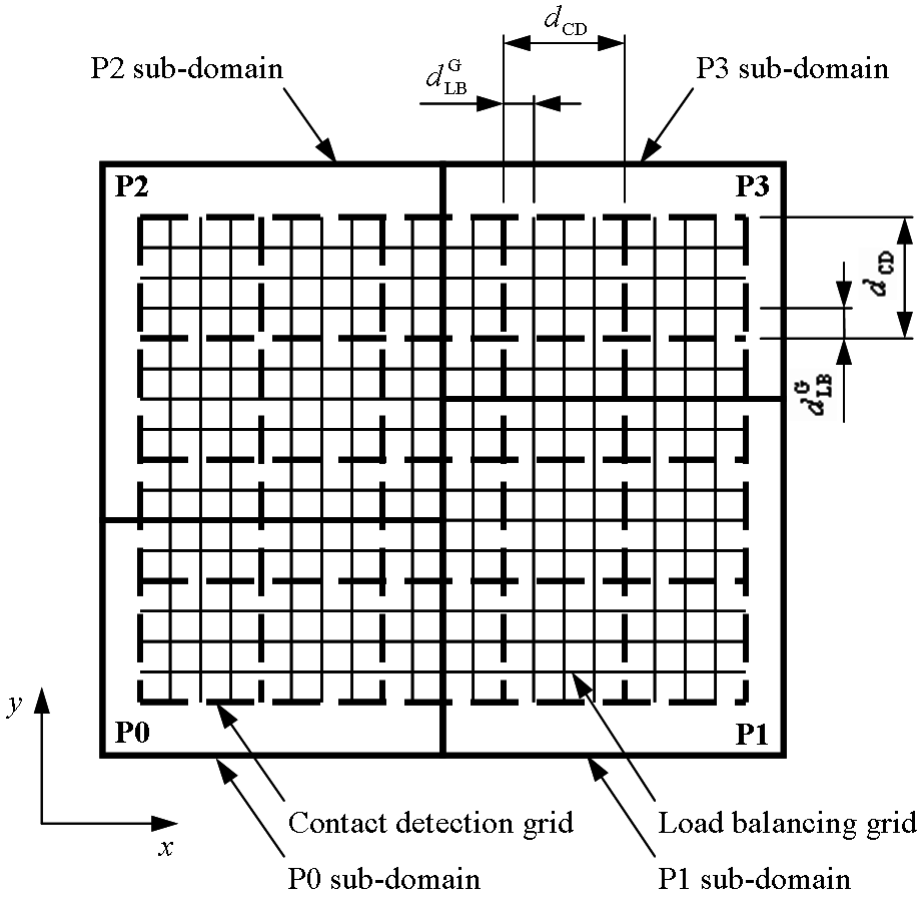


Figure 3.47: Global contact detection grid overlaid by the global load balancing grid.

The size of maximum imbalance B_{max} has a big impact on the performance of the parallel implementation and must be carefully chosen, see Chapter 4 for performance

tests.

The LB grid is assembled by using the contact detection (CD) grid. The size of each cell in CD grid is equal to the maximum element diameter d_{CD} (Figure 3.47) and then the size of each cell in LB grid d_{LB} is calculated as follows:

$$d_{LB} = \frac{d_{CD}}{I_{LB}} \tag{3.8}$$

where I_{LB} is a parameter bigger than 1 specified in the input file. Hence, the resolution of the LB grid is finer than the resolution of the CD grid in order to achieve better re-partitioning. Figure 3.47 shows an example of both grids (global) on four processors.

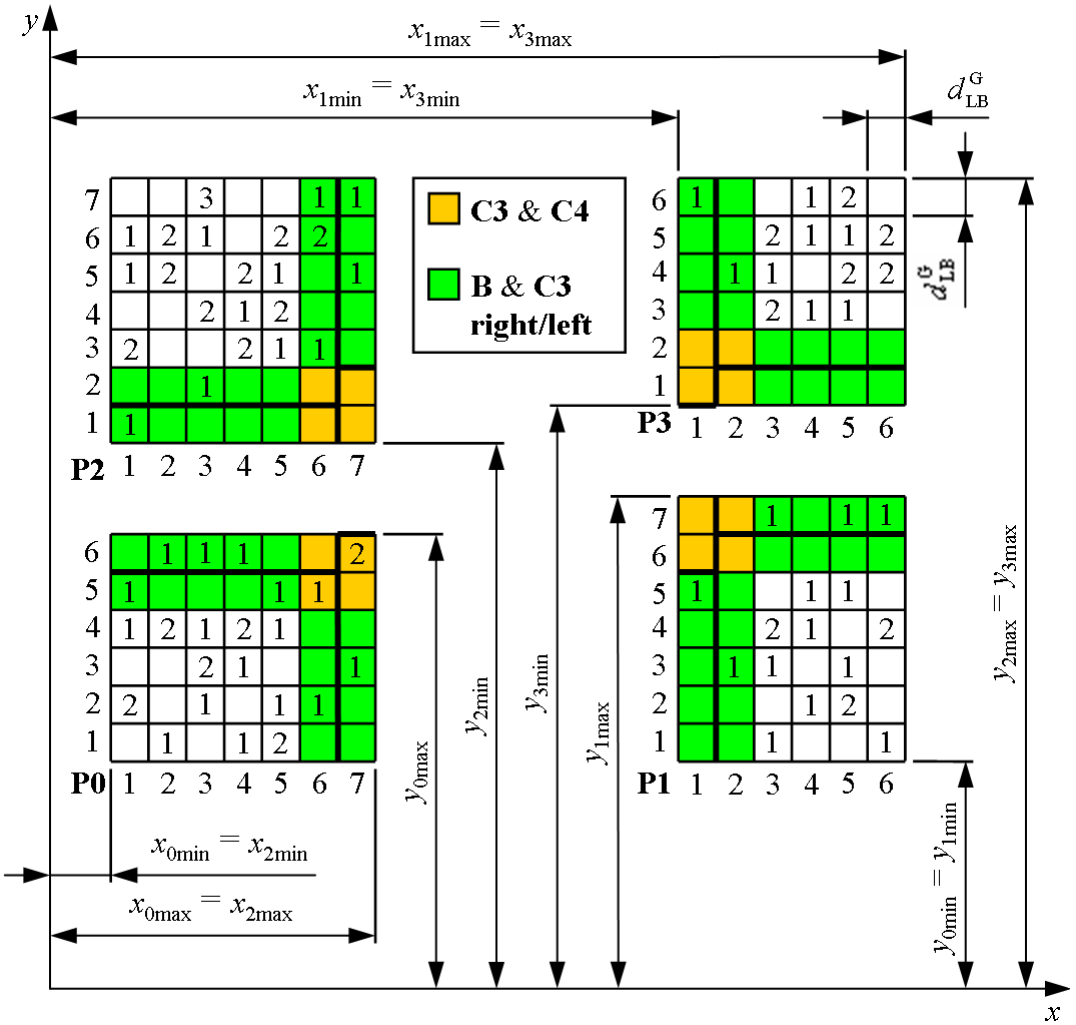


Figure 3.48: Local load balancing grid assembled on four processors. Each cell contains count of elements located within the cell.

As mentioned in Chapter 3.9, the contact detection search is performed locally on

each processor. This means that the LB grid is also assembled locally, see Figure 3.48. Elements located within each cell in the LB grid are saved in a separate list and the count of these elements is saved as well. In order to estimate the workload in each cell better, the count of elements in each cell is increased by the number of contacting couples for each element located within the cell. Then counts of elements within each cell of the local LB grid on each processor (Figure 3.48) are gathered on all processors by using the MPI_Allgather() communication function and the global LB grid is assembled, see Figure 3.49. As a result the copy of the global LB grid (counts only) is saved on each processor.

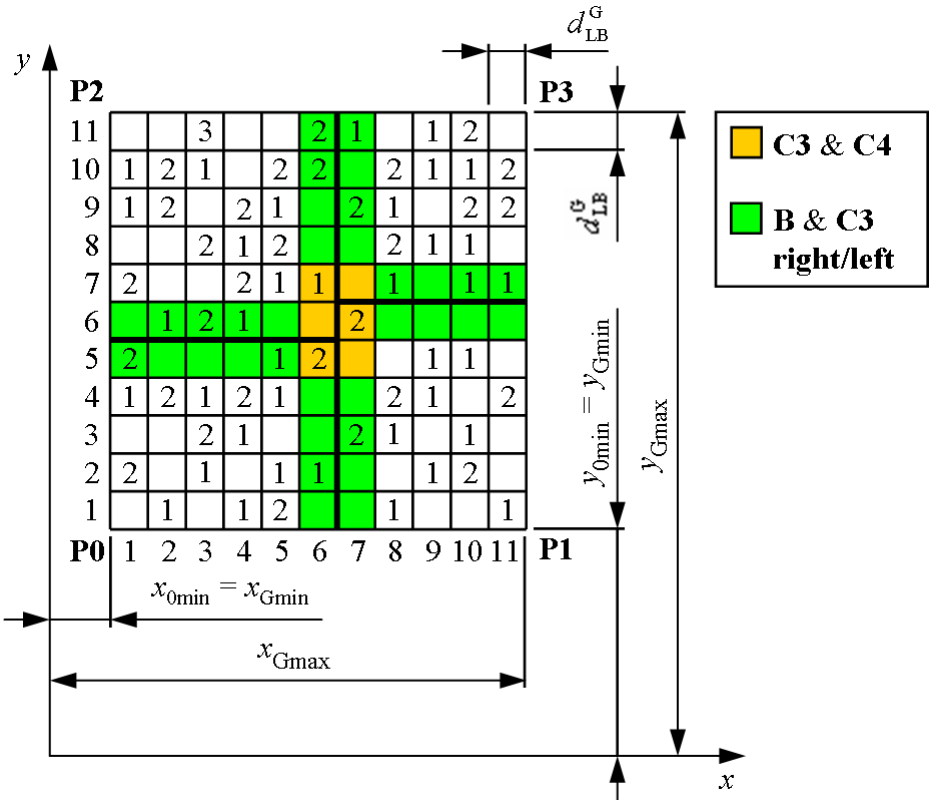


Figure 3.49: Global LB grid assembled by combining local LB grids from Figure 3.48. Counts in coloured cells are added up.

Step 2. Once the global LB grid (counts only) is assembled on each processor, re-partitioning is performed. The partitioning is done in the following 4 steps:

1. Sums of each column in global LB grid are calculated, see Figure 3.50.

2. Computational domain is partitioned in x direction to the specified number of columns *icol* (see Figure 3.50) by using Algorithm 3.1 (see Chapter 3.5 for details). It is worth noting that the border of the sub-domain is moved in units for

which one unit is equal to the size of one cell d_{LB} in LB grid, see Figure 3.50. Hence, a finer LB grid means better re-partitioning resulting in a very small imbalance.

3. Sums of each row in the global LB grid are calculated for each column separately, see Figure 3.51.

4. Each column is partitioned in y direction to the specified number of rows *irow*, see Figure 3.51. Each column is partitioned separately by using Algorithm 3.1 (x coordinate is replaced by y coordinate).

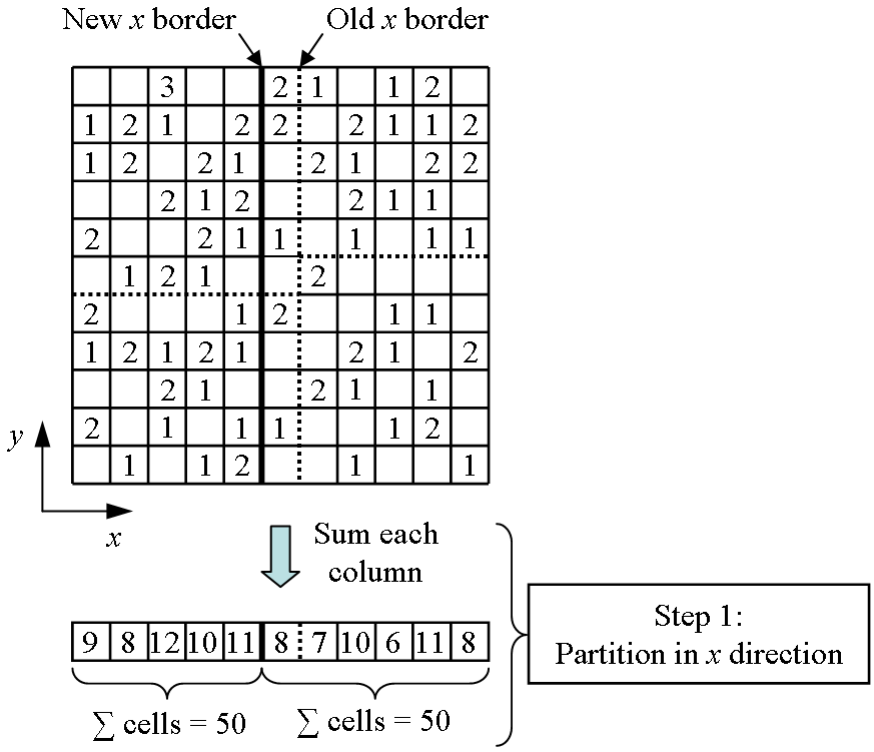


Figure 3.50: Partitioning of global LB grid in x direction.

Partitioning of the global LB grid from Figure 3.49 in x and y direction is illustrated in Figures 3.50 and 3.51.

Step 3. Since the grid of processors has changed due to re-partitioning, lists of neighbouring processors and of communication couples at left and right borders must be updated. Neighbouring processors are saved in a 2D integer array **i2neigh** (Figure 3.53) of size *irow* (number of rows in the grid of processors), since any processor cannot have more than *irow* neighbours at a vertical border. Number of segment *isegm*

is saved for each neighbouring processor in a 2D integer array **i2segm** of the same size. Lastly, coordinates of neighbouring borders located above bottom border and below top border of a current processor are saved in a 2D floating point array **d2neigh** of size *irow*.

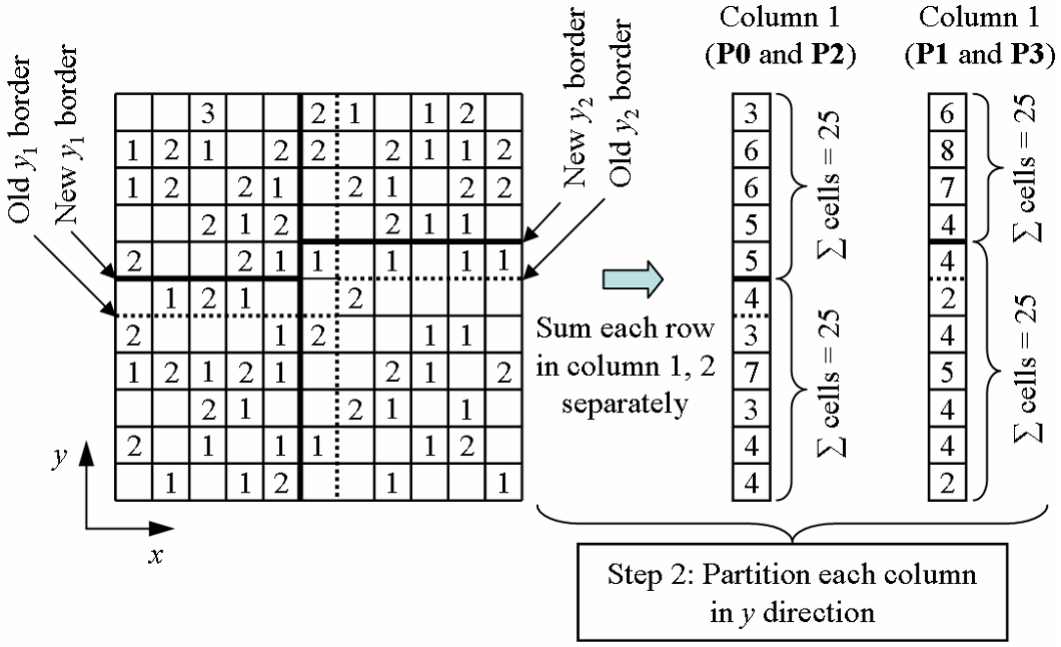


Figure 3.51: Partitioning of global LB grid in *y* direction.

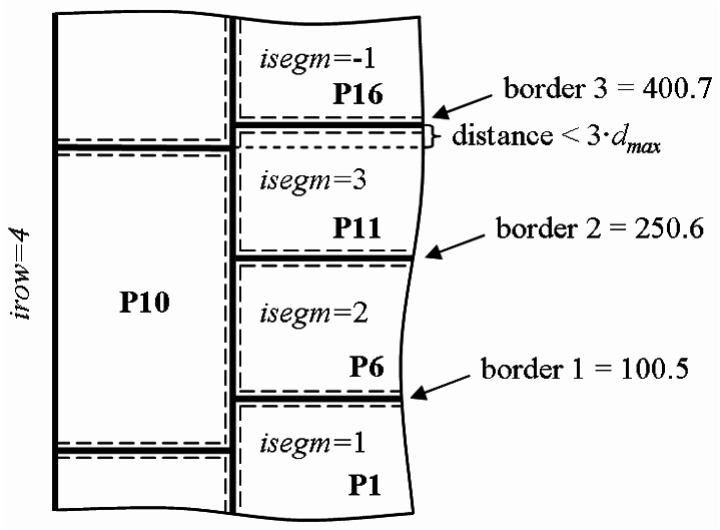


Figure 3.52: Neighbouring processors of processor 10.

The list of communication couples is assembled by using the list of neighbouring processors, for details refer to Chapter 3.12.

Figure 3.53 shows a list of neighbouring processors at right border assembled for processor 10 (see Figure 3.52).

Neighbours for Processor 10:

Integer array **i1neigh** [2]	4	Count of neighbours on the right.
	-1	Count of neighbours on the left.
Integer array **i2neigh** [irow]	1 6 11 16	List of neighbours on the right.
	-1 -1 -1 -1	List of neighbours on the left.
Integer array **i2segm** [irow]	1 2 3 -1	*isegm* for each neighbour (right).
	-1 -1 -1 -1	*isegm* for each neighbour (left).
Floating point array **d2neigh** [irow]	100.5 250.6 400.7 0.0	Border coordinates (right).
	0.0 0.0 0.0 0.0	Border coordinates (left).

Figure 3.53: List of neighbouring processors of processor 10.

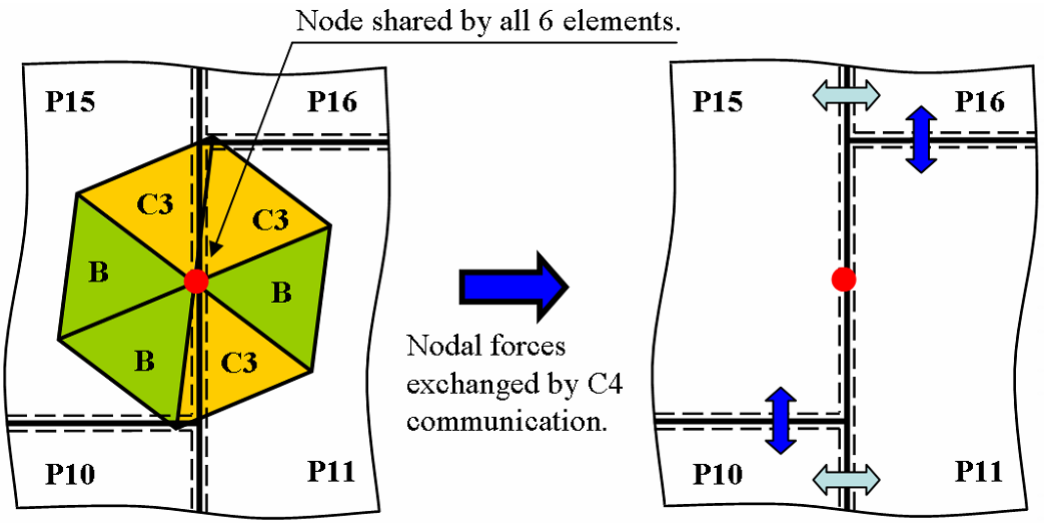


Figure 3.54: Node shared by 2 C3 elements located on processors 10 and 16.

Processor 16 is saved to the list of neighbours (Figure 3.53) even though it is not directly in contact with processor 10, but the distance from top border of processor 10 to the bottom border of processor 16 is less than $3d_{max}$ where d_{max} is a maximum element diameter. Processor 16 must be included in case a node is shared by two C3 elements from which one is located on processor 10 and second is located on processor

16, see Figure 3.54. Then this node cannot be sent in an usual C3 communication (Figure 3.66c) since the node is located on all 4 processors.

Algorithm 3.7 summarises the search for neighbouring processors at the right border in a form of a pseudo-code. The left border is processed analogically to the right border. Since the re-partitioning of the computational domain is done on all processors, each processor has, in its memory, coordinates of borders of all sub-domains in the grid. Therefore, it is possible to perform the search for neighbouring processors straight away.

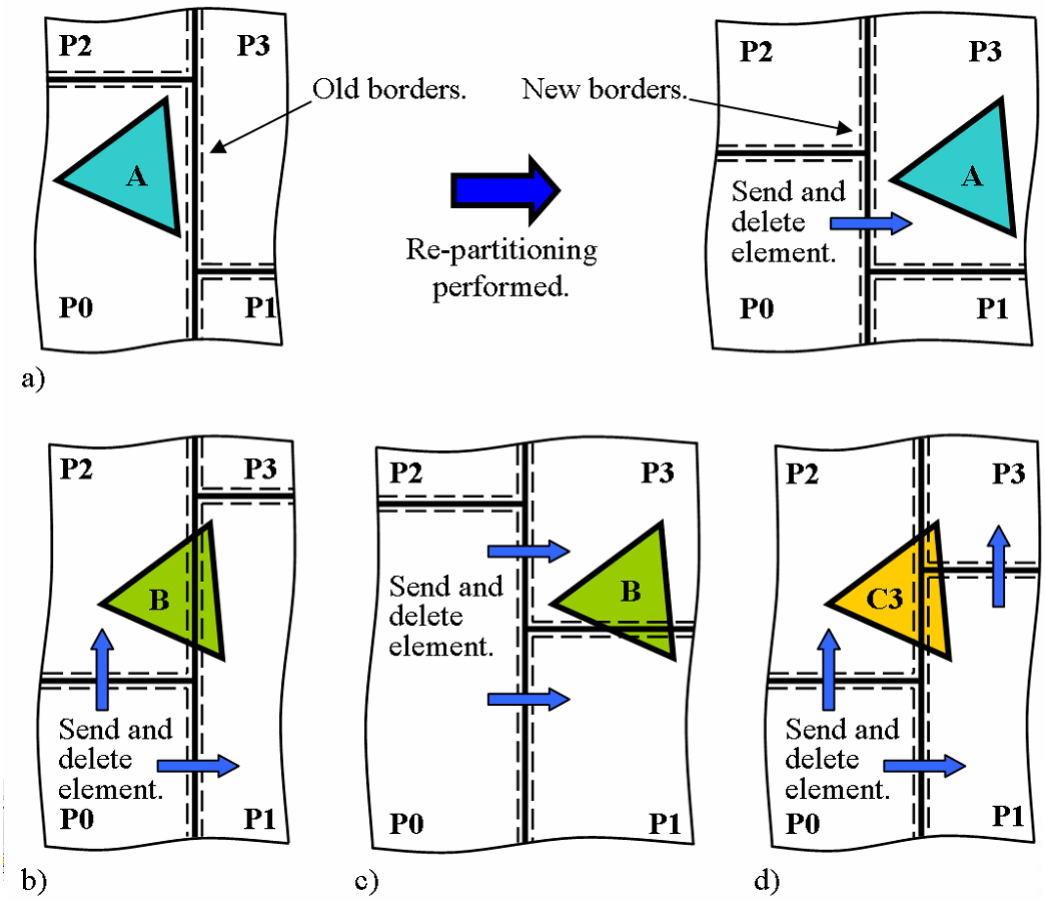


Figure 3.55: a) Internal element A becomes internal element A on neighbouring processor, b) and c) Element becomes interfacial element B on two neighbouring processors, d) Element becomes interfacial element C3 on 3 neighbouring processors.

Algorithm 3.7 Search for neighbouring processors at right border.

```
 1: double irow, icol;                          ▷ Number of rows/columns in the grid.
 2: double nir, nic;                  ▷ Number in row/column of the current processor.
 3: double dcbuff;                      ▷ Buffer zone around borders of sub-domain.
 4: rcol = nir + 1;                          ▷ Number of neighbouring right column.
 5: diam = 3 · diam;                          ▷ Maximum diameter of element * 3.
 6: dy1 = bottom border − dcbuff/2 − diam;                ▷ Expand bottom border.
 7: dy2 = top border + dcbuff/2 + diam;  ▷ Expand top border of current processor.
 8: if (rcol < icol) then          ▷ Exclude the last column in the grid of processors.
 9:     for (i = 0; i < (irow − 1); i++) do          ▷ Neighbouring processors (right).
10:         procn = rcol + (i · icol);                          ▷ Rank of the neighbour.
11:         dny1 = bottom border of the neighbour;
12:         dny2 = top border of the neighbour;
13:         icount = i1neigh[0];                                ▷ Count of neighbours.
14:         if ((nic == 0)&&(i == 0)) then              ▷ 1st processors in columns.
15:             save neighbour; save segment; increase count;          ▷ See Figure 3.53.
16:             if ((dny2 − dcbuff/2) < dy2) then                ▷ dny2 is below dy2.
17:                 procn = procn + icol;              ▷ Next neighbour in right column.
18:                 save neighbour (procn); save border (dny2); increase count by 2;
19:                 if (dny2 < top border of current processor (exact)) then
20:                     save seigment;                          ▷ Array i2segm (Figure 3.53).
21:                 end if
22:             end if
23:         else if ((dy1 < (dny2 + dcbuff/2))&&((dny2 − dcbuff/2) < dy2)) then
                ▷ dny2 is above bottom and below top border of current processor.
24:             if (icount > 0) then
25:                 icount = icount − 1;
26:             end if
27:             save neighbour (procn);                         ▷ Array i2neigh (Figure 3.53).
28:             if (bottom border of current processor (exact) < dny2) then
29:                 save seigment;                              ▷ Array i2segm (Figure 3.53).
30:             end if
31:             procn = procn + icol;              ▷ Next neighbour in right column.
32:             save neighbour; save border (dny2); increase count by 2;▷ Figure 3.53.
33:             if (dny2 < top border of current processor (exact)) then
34:                 save next seigment;                         ▷ Array i2segm (Figure 3.53).
35:             end if
36:         else if (((dny1 + dcbuff/2) < dy1)&&(dy2 < (dny2 − dcbuff/2))) then
                ▷ Current processor is whole inside neighbour => one neighbour only.
37:             save neighbour; save segment; increase count;          ▷ See Figure 3.53.
38:         else if (i == (irow − 2)) then  ▷ Next to last neighbour in right column.
39:             if (icount == 0) then                           ▷ No neighbour saved yet.
40:                 procn = procn + icol;▷ Rank of the last neighbour in right column.
41:                 save neighbour; save segment; increase count;     ▷ See Figure 3.53.
42:             end if
43:         end if
44:     end for
45: end if
```

Step 4. In order to reflect changed positions of borders in the grid of processors, the position of elements located in the proximity of borders of each sub-domain must be checked and elements must be re-distributed among processors. Since the change in size of each sub-domain is small due to the fact that RCB algorithm is an incremental partitioner, only elements located close to borders must be re-distributed, which is only a fraction of the total number of elements within the sub-domain. This results in a significant reduction of the cost of load balancing compared with non-incremental partitioners.

The process of re-distribution of elements is analogical to the migration of elements (see Chapter 3.10) with a few important differences:

- The positions of elements against new borders is checked directly by using cells in the LB grid located in the proximity of borders, unlike the moving of elements where the position of elements is checked by using singly connected lists (see Figure 3.28) assembled during contact detection.

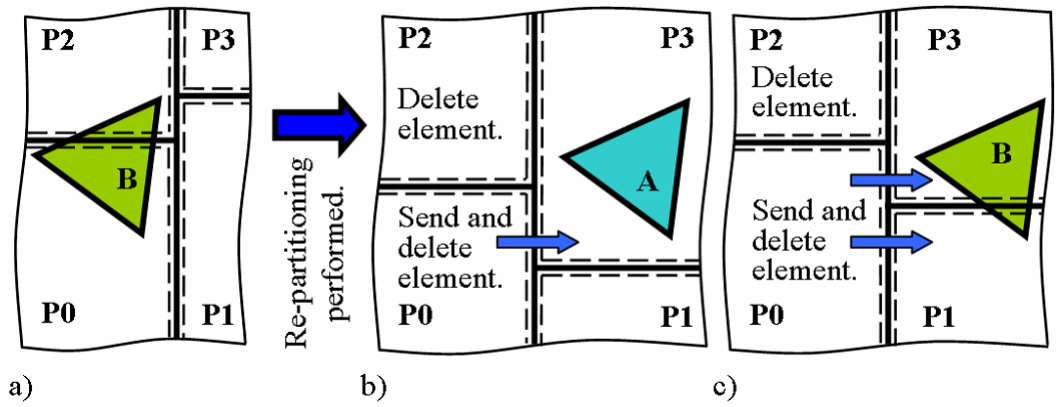


Figure 3.56: a) Original position of interfacial element B within sub-domain, b) Element becomes internal element A on neighbouring processor, c) Element becomes interfacial element B on two neighbouring processors.

- The distance between old and new positions of borders is limited only by the maximum imbalance specified in the input file, unlike the moving of elements where the maximum travelled distance is limited by the size of the buffer around borders of the sub-domain which, in turn limits the number of possible combinations of changes in status of the element. Therefore, some status changes are impossible during moving of elements, unlike the re-distribution of elements where all possible combinations must be taken into account while checking position of elements, for instance:

– Internal element A can leave sub-domain and become internal element A (Figure 3.55a) or interfacial element B (Figure 3.55b and c) or C3 (Figure 3.55d) on neighbouring processor(s).

– Interfacial element B located at horizontal border can leave both processors and become internal element A (Figure 3.56b) or interfacial element B (Figure 3.56c) or C3 on neighbouring processor(s).

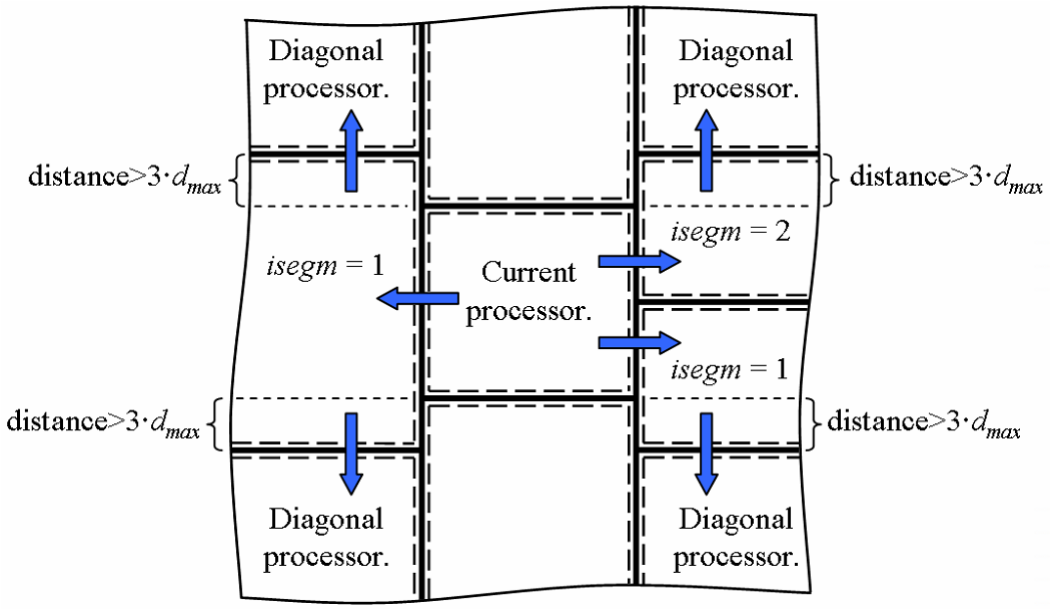


Figure 3.57: Communication pattern for diagonal processors.

Since the distance by which borders move can be bigger than $3 \cdot diam$, where $diam$ is a maximum diameter of the element, some element must be sent to a processor which is not saved in the list of neighbouring processors, see Figure 3.55d. In that case, the element is first sent in a horizontal message to the last neighbouring processor and then sent again from the receiving processor in a vertical message. The communication pattern for diagonal processors at all corners is in Figure 3.57.

3.12 Communication

Since all processors are assigned a sub-domain during a domain decomposition, it is necessary to coordinate computations on each processor by explicitly sending and receiving messages. Communication among processors is performed in the following cases:

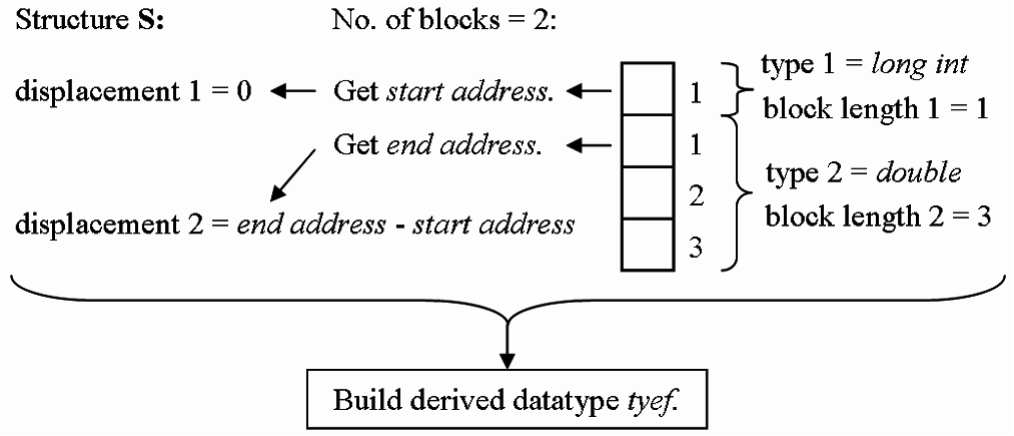


Figure 3.58: Information used to build a derived datatype *tyef* from a structure **S**.

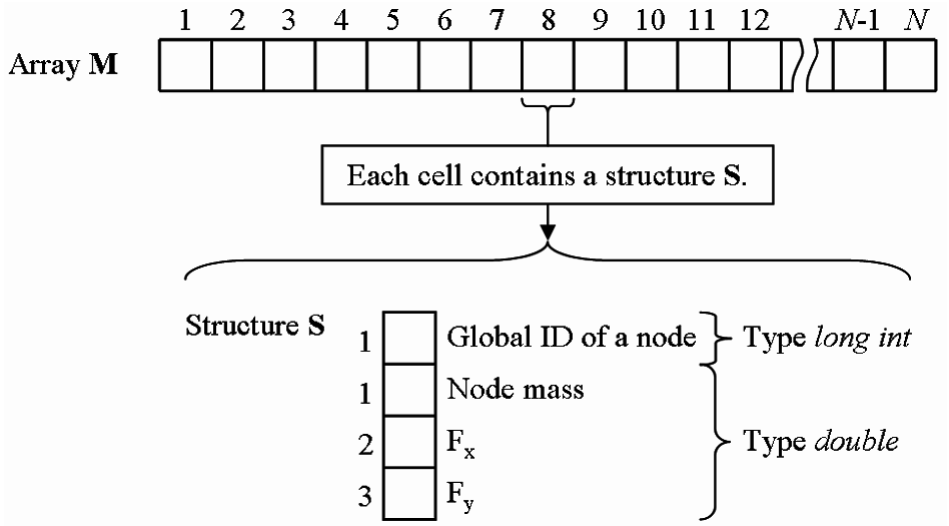


Figure 3.59: Data structure of a message for nodal forces exchange: message is saved in an array **M** where each cell contains a structure **S**.

- Nodal forces exchange: Nodal forces and mass of nodes belonging to elements located at borders between sub-domains must be exchanged in every time step and summed up. Messages for this communication are assembled by using singly connected lists of elements saved in arrays **H** and **L**, see Figures 3.27 and 3.28. The message itself is saved in an array **M** of size N where each cell contains a structure **S** of one integer number (type *long int*) and three floating point numbers (type *double*), see Figure 3.59. Each cell in the array **M** holds information for one node (global ID, node mass and forces in x and y direction). For this structure a derived datatype *tyef* is built during the first force exchange by using information from the Figure 3.58, i.e. number of blocks, block length,

type and displacement of each block.

- Moving of elements: Elements are moved only if the maximum travelled distance is greater or equal to the size of a buffer zone (see Chapter 3.4.1). The structure of the message as well as building of a derived datatype for it is described in detail in Chapter 3.10.2.

- Load balancing: If the level of imbalance in the grid of processors reaches prescribed maximum value, load balancing is triggered. When re-partitioning is done, elements are redistributed among processors. The structure of the message is therefore identical to the message needed during moving of elements since both messages contain elements and their nodes.

- Collective communication: Unlike the previous three cases this communication is among all processors in the grid and it is needed to:

  - gather a maximum travelled distance from each processor and place a copy on each processor in order to check if moving of elements should be triggered. A MPI function for collective communication MPI_Allgather() (see Chapter 2.4.2) is used. This communication is performed in every time step.
  - gather workload from each processor and save a copy on each processor by using MPI_Allgather() function in order to control when load balancing should be performed. Workload is communicated only if count of moving of elements is equal to prescribed value.
  - when load balancing is performed, it is necessary to gather content of local load balancing cells on each processor and combine it into a global load balancing grid (see Chapter 3.11). Function MPI_Allgather() is used for this communication.
  - gather counts of elements which must be written into output on each processor by using MPI_Allgather() function. Frequency of this communication is equal to the frequency of output, see Chapter 3.13.

In order to send and receive messages in the first three cases (force exchange, migration of elements, redistribution of elements during load balancing), some available communication engine could be employed, for instance an engine introduced by Munjiza, et al.[122,167] In this engine, communication couples are assembled by performing contact

detection among neighbouring processors. All communication couples are divided into time slots and each processor is allowed to communicate with only one neighbouring processor in one time slot. The downside of this approach is that not all processors are participating in communication in each time slot, for instance all communication for a grid of 14 processors in Figure 3.60 would be finished in 8 time slots since processor 5 has to separately communicate with 8 different neighbouring processors.

The communication engine used in this work employs a different approach, since assembling communication couples by using a contact detection search, gives no control over the direction of the communication. An important assumption for both engines is that the maximum size of any discrete element is much smaller than the size of any sub-domain. Therefore, each processor has to communicate only with neighbouring processors which are in direct contact with its sub-domain.

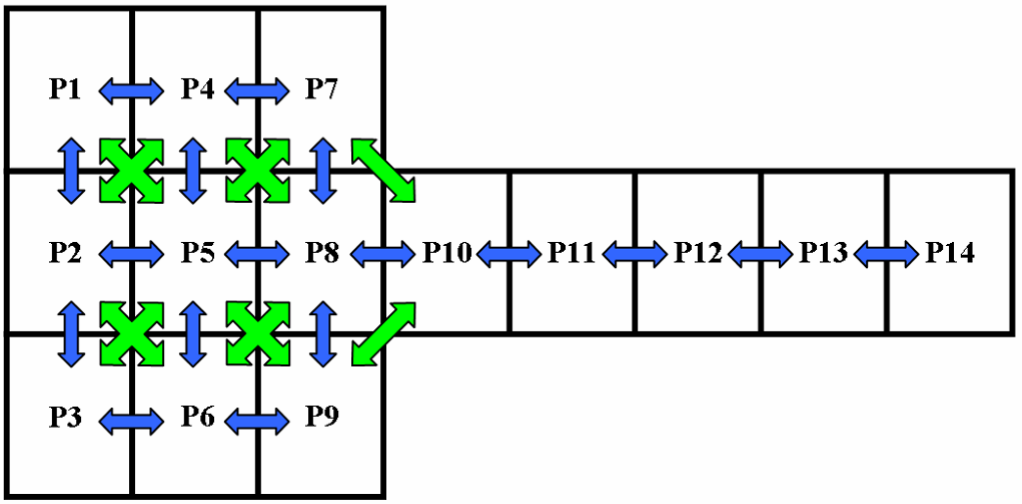


Figure 3.60: Communication couples in the grid of 14 processors. Figure adapted from Munjiza, et al.[122]

Communication is performed in two separate stages: horizontal and vertical communication. In the first stage horizontal messages are assembled (for right and left borders) and exchanged. In the second stage vertical messages are assembled (for top and bottom borders) and exchanged in vertical direction. If a communication with a neighbouring diagonal processor is needed then the information for the diagonal processor is first sent in the horizontal message and then the receiving processor sends it again in the vertical message. Therefore no direct diagonal communication is performed and the communication for the grid in Figure 3.60 would be finished in 4 time slots (blue arrows only).

It is worth noting, that the amount of data needed for communication in a diagonal direction (elements located at a corner) is very small compared with communication in a horizontal or vertical direction (elements located at a border). Initiating a separate communication with a diagonal neighbouring processor would be, therefore, more expensive in terms of CPU time than the slight increase in cost of horizontal and vertical communication.

An MPI communication function MPI_Sendrecv() (see Chapter 2.4.2) is used to exchange messages among neighbouring processors. This function is a so-called "blocking" communication function which means the receiving processor has to wait until a message is received before starting another communication with a different processor. If, for instance, horizontal communication was initiated on all processors at the same time (see Figure 3.61), messages between processors would be exchanged in a wave-like fashion, since processors with higher rank would have to wait until all messages between processors with lower rank are exchanged, e.g. processor 4 in Figure 3.61 would have to wait until communication among processors 0-1, 1-2 and 2-3 is finished before receiving message from processor 3. This would slow down communication significantly, resulting in higher CPU time.

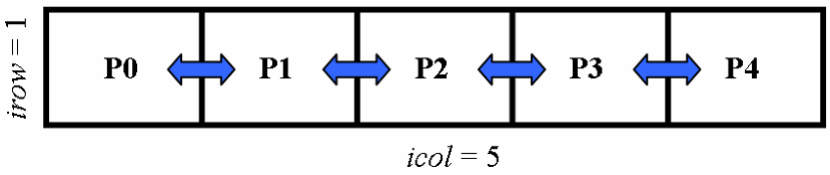


Figure 3.61: Horizontal messages exchanged in the grid of five processors.

For this reason horizontal and vertical communication is performed in two steps:

1. Exchange messages among communication couples, formed starting from the bottom-left corner of the grid of processor, i.e processor 0:

   (a) Communication couples for horizontal communication are formed among processors in two neighbouring columns with number in row (*nir*) equal to 0-1, 2-3, etc., see Figure3.62. For all processors in the same column *nir* has the same value.

   (b) Communication couples for vertical communication are formed among processors with number in column (*nic*) equal to 0-1, 2-3, etc., see Figure 3.62.

2. Exchange messages among remaining communication couples, i.e. *nir* and *nic* for horizontal and vertical communication are equal to 1-2, 3-4, etc., see Figure 3.62.

Communication is initiated on all processors in corresponding columns/rows at the same time. As a result horizontal/vertical messages are exchanged in only two steps regardless of the size of grid of processors (number of processors).

Number in row *nir* and number in column *nic* are calculated as follows

$$\begin{aligned} nir &= rank\%icol \\ nic &= rank/icol \end{aligned} \tag{3.9}$$

where *rank* is number assigned to each processor by MPI at the start of the simulation and *icol* is a total number of columns in the grid of processors, see Figure 3.62. Operators "%" and "/" have a C++ meaning for operations on integer numbers, i.e. remainder after division and division by integer number respectively.

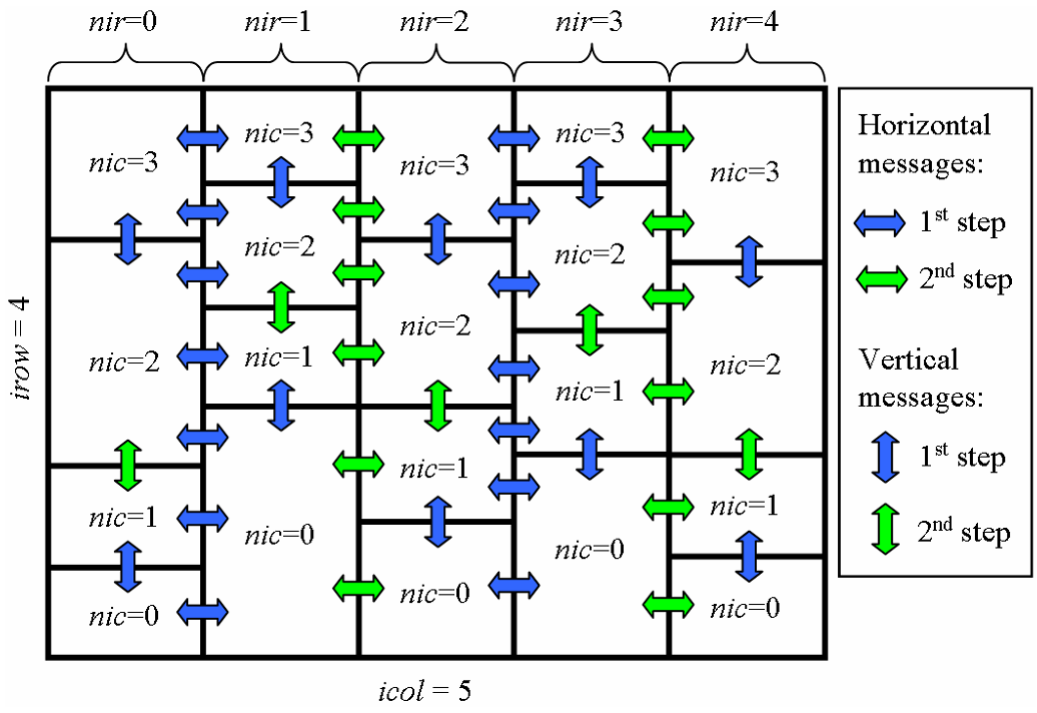


Figure 3.62: Horizontal and vertical messages are exchanged in two steps: 1) messages are exchanged between processors in rows/columns equal to 0-1, 2-3, etc.; 2) messages are exchanged between processors in rows/columns equal to 1-2, 3-4, etc.

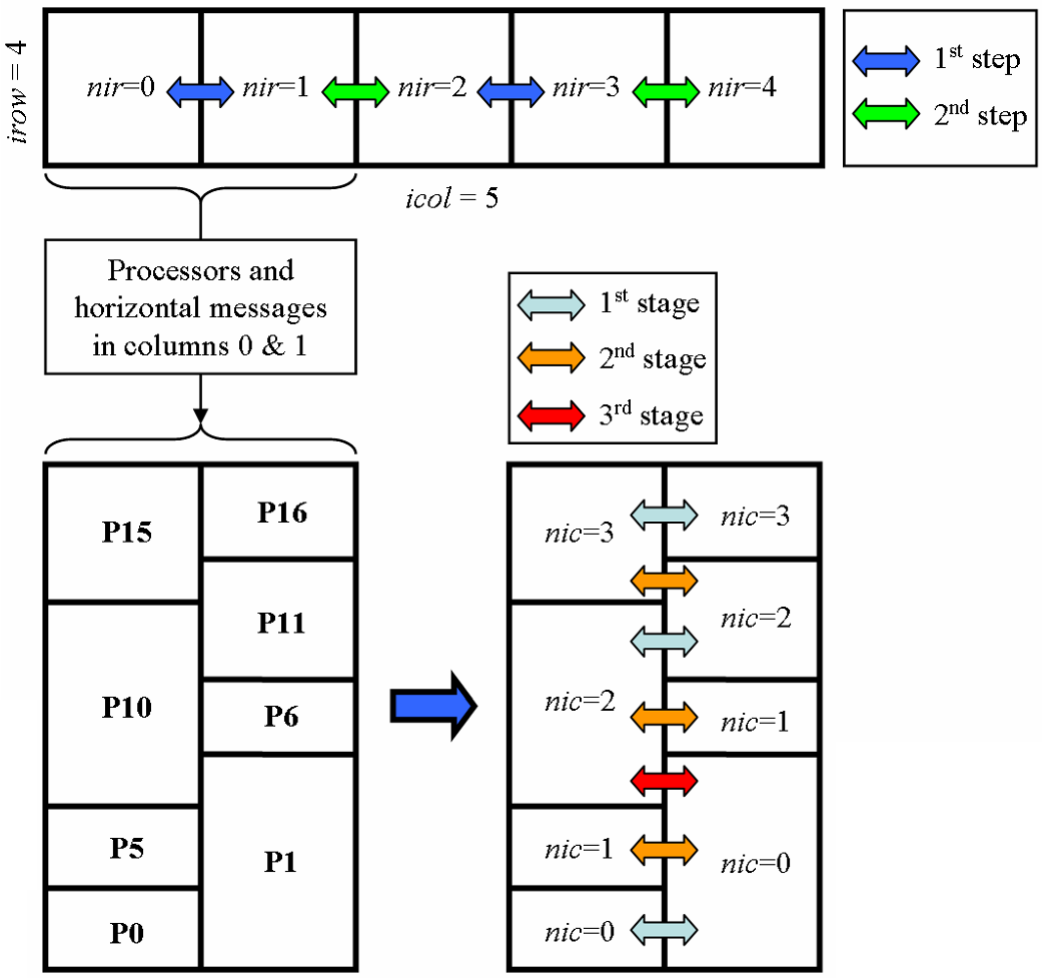


Figure 3.63: Horizontal messages are exchanged in several stages: 1) messages are exchanged between processors with equal *nic*; 2) messages are exchanged between processors with $nic \pm 1$; 3) messages are exchanged between processors with $nic \pm 2$.

Communication couples for both steps don't have to be prepared in advance, since number in row *nir* and number in column *nic* can be used directly, see Algorithm 3.9. Vertical communication is finished in just two steps, described above, since each processor has only one neighbouring processor at each horizontal border. Unlike vertical communication, both steps during horizontal communication must be performed in several stages because each processor can have more than one neighbouring processor at each vertical border. For instance columns 0 and 1 in Figure 3.62 have four processors each and if each processor had only one neighbour then only four messages exchanged at the same time would be needed to complete the communication. In order to complete the horizontal communication between columns 0 and 1 (see Figure 3.62) seven messages must be exchanged. If all these messages were sent at the same

time, it would again be performed in a wave-like fashion, since processors with higher rank would have to wait until all messages between processors with lower rank are exchanged. This would, again, have a big impact on the communication's performance.

Therefore, the horizontal communication between two neighbouring columns of processors (Figure 3.63) must be done in several stages to avoid interlocking. The number in column *nic* on each processor is used to assemble communication pairs for horizontal communication. Messages between processors with equal *nic* are exchanged during the first stage, see Figure 3.63. In the second stage, communication between processors with $nic \pm 1$ is performed; in the third stage, messages between processors with $nic \pm 2$ are exchanged, etc.

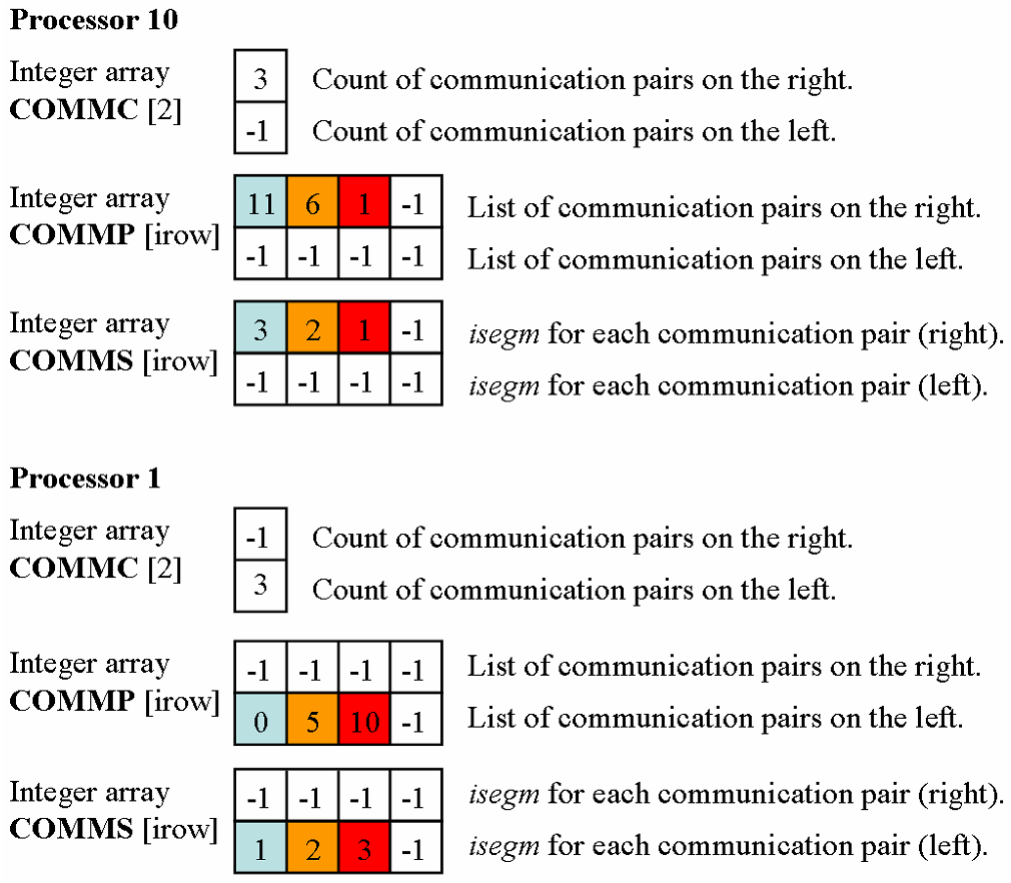


Figure 3.64: Communication pairs for processors 10 and 1 from Figure 3.63.

Communication couples for horizontal communication are assembled by using lists of neighbouring processors for the right and left border, see Chapter 3.11 for details. These lists, together with communication couples, are created when the domain decomposition at the start of the simulation is performed and then updated during each load balancing. Communication pairs are saved in an 2D integer array **COMMP** (Fig-

ure 3.64) of size *irow* (number of rows in the grid of processors) since any processor cannot have more than *irow* neighbours at vertical border. It is important to also save a segment *isegm* (Figure 3.64) for each pair, in order to directly access a proper list of elements/nodes designated for sending to the segment (see Figures 3.28 and 3.35). For instance, communication pairs for processors 1 and 10 in Figure 3.63 would be 0, 5, 10 and 11, 6, 1 respectively, see Figure 3.64. Corresponding segments for processors 1 and 10 are marked in the Figure 3.65.

Processors 11 and 15 have only 2 communication pairs and the remaining processors have only 1. Thus all the communication between processors in Figure 3.63 would be finished in three separate stages.

The process of assembling communication couples for horizontal communication is summarised in Algorithm 3.8.

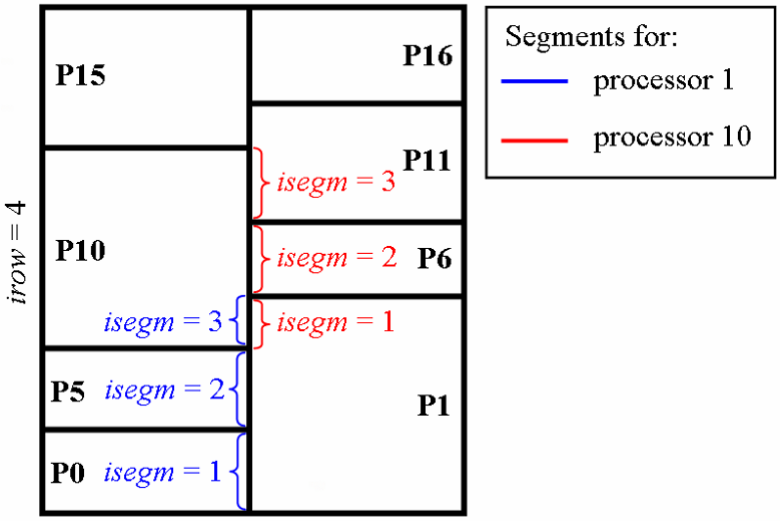


Figure 3.65: Segments (neighbouring processors) for processors 10 and 1 from Figure 3.63.

The whole communication engine used in this work is summarised in the form of a pseudo-code in Algorithm 3.9. The main advantage of this approach is to have a control over communication order in the sense that the order of each message is set and the communication engine is, therefore, easier to implement. Also, by eliminating messages in diagonal direction the whole communication is finished in fewer time slots than the in the case of the engine introduced by Munjiza, et al.[122,167] This reduces the CPU time since each communication has a network latency, which is the time required to establish a connection between two processors.

Algorithm 3.8 Assembling horizontal communication pairs.

```
 1: integer nic;                                   ▷ Number in column (current processor).
 2: integer irow, col;                             ▷ Number of rows/columns in the grid of processors.
 3: integer i, j, k, m, stage;
 4: integer i2nic;                                 ▷ To store nic of neighbouring processors.
 5: integer *i1neigh;                              ▷ Number of right/left neighbouring processors.
 6: integer **i2neigh;                             ▷ Ranks of right/left neighbouring processors.
 7: integer **i2segm;                              ▷ Segments for right/left neighbouring processors.
 8:
 9: Delete old communication pairs;
10: for (i = 0; i < 2; i++) do                     ▷ Prepare i2nic and 1st stage (equal nic).
11:     for (j = 0; j < i1neigh[i]; j++) do        ▷ Number of right/left neighbours.
12:         if (i2segm[i][j] > −1) then            ▷ If message needed.
13:             i2nic[i][j] = i2neigh[i][j]/icol;  ▷ Rank of neighbour / icol.
14:             if (nic == i2nic[i][i]) then       ▷ If 1st stage of horizontal communication.
15:                 COMMP[i][COMMC[i]] = i2neigh[i][j];   ▷ Rank of neighbour.
16:                 COMMS[i][COMMC[i]] = i2segm[i][j];    ▷ Save segment.
17:                 COMMC[i] = COMMC[i] + 1;              ▷ Increase count.
18:                 Mark neighbour as processed;
19:             end if
20:         end if
21:     end for
22: end for
23: for (stage = 1; stage < irow; stage++) do      ▷ 2nd, 3rd,... stage.
24:     for (k = 0; k < 2; k++) do                 ▷ Each stage is either + or -.
25:         if (k == 0) then
26:             m = nic + stage;
27:         else
28:             m = nic − stage;
29:         end if
30:         for (i = 0; i < 2; i++) do             ▷ Right = 0, left = 1.
31:             for (j = 0; j < i1neigh[i]; j++) do   ▷ No. of right/left neighbours.
32:                 if (i2segm[i][j] > −1)&&(neighbour not processed) then
33:                     if (m == i2nic[i][i]) then    ▷ If 2nd, 3rd,... stage.
34:                         COMMP[i][COMMC[i]] = i2neigh[i][j];   ▷ Save rank.
35:                         COMMS[i][COMMC[i]] = i2segm[i][j];    ▷ Save segment.
36:                         COMMC[i] = COMMC[i] + 1;              ▷ Increase count.
37:                         Mark neighbour as processed;
38:                     end if
39:                 end if
40:             end for
41:         end for
42:     end for
43: end for
```

Algorithm 3.9 Horizontal and vertical communication.

```
1: integer nir, nic;                                   ▷ Number in row, number in column.
2: integer icol;                                       ▷ Number of columns in the grid of processors.
3: integer idest, isour;                               ▷ Message destination, source.
4: integer i, j, right, left, top, bottom, ibor;
5: for (i = 0; i < 2; i++) do                          ▷ Horizontal messages sent in two steps.
6:     ibor = 1;                                       ▷ Set ibor for left messages.
7:     if (i == 0) then                                ▷ First step.
8:         left = 1; right = 0;
9:     else                                            ▷ Second step (i = 1).
10:        left = 0; right = 1;
11:    end if
12:    if (nir%2 == left)&&(nir > 0) then              ▷ Send/receive left.
13:        for (j = 0; j < COMMC[ibor]; j++) do        ▷ See Figure 3.64.
14:            idest = COMMP[ibor][j]; isour = idest;
15:            Prepare left message for isegm;         ▷ Use isegm saved for each pair.
16:            MPI_Sendrecv(message, idest, isour);    ▷ Send/receive left message.
17:            Save content of the message into database;
18:        end for
19:    end if
20:    ibor = 0;                                       ▷ Set ibor for right messages.
21:    if (nir%2 == right)&&(nir < (icol − 1)) then    ▷ Send/receive right.
22:        for (j = 0; j < COMMC[ibor]; j++) do        ▷ See Figure 3.64.
23:            idest = COMMP[ibor][j]; isour = idest;
24:            Prepare right message for isegm;        ▷ Use isegm saved for each pair.
25:            MPI_Sendrecv(message, idest, isour);    ▷ Send/receive right message.
26:            Save content of the message into database;
27:        end for
28:    end if
29: end for
30: for (i = 0; i < 2; i++) do                         ▷ Vertical messages sent in two steps.
31:    if (i == 0) then                                ▷ First step.
32:        bottom = 1; top = 0;
33:    else                                            ▷ Second step (i = 1).
34:        bottom = 0; top = 1;
35:    end if
36:    Prepare top and bottom messages;                ▷ One message for each direction.
37:    if (nic%2 == bottom)&&(nic > 0) then            ▷ Send/receive bottom.
38:        idest = rank − icol; isour = idest;
39:        MPI_Sendrecv(message, idest, isour);
40:    end if
41:    if (nic%2 == top)&&(nic < (irow − 1)) then      ▷ Send/receive top.
42:        idest = rank + icol; isour = idest;
43:        MPI_Sendrecv(message, idest, isour);
44:    end if
45:    Save content of both messages into database;
46: end for
```

For an Ethernet network the network latency is approximately 1 ms. The time needed to send a message through the network can be calculated as follows:[91]

$$T_m = T_0 + T_b B \tag{3.10}$$

where T_m is the time needed to send a message of size equal to B bytes, T_0 is the network latency and T_b is the time required to send one byte of data. Thus the efficiency of communication improves with larger messages.

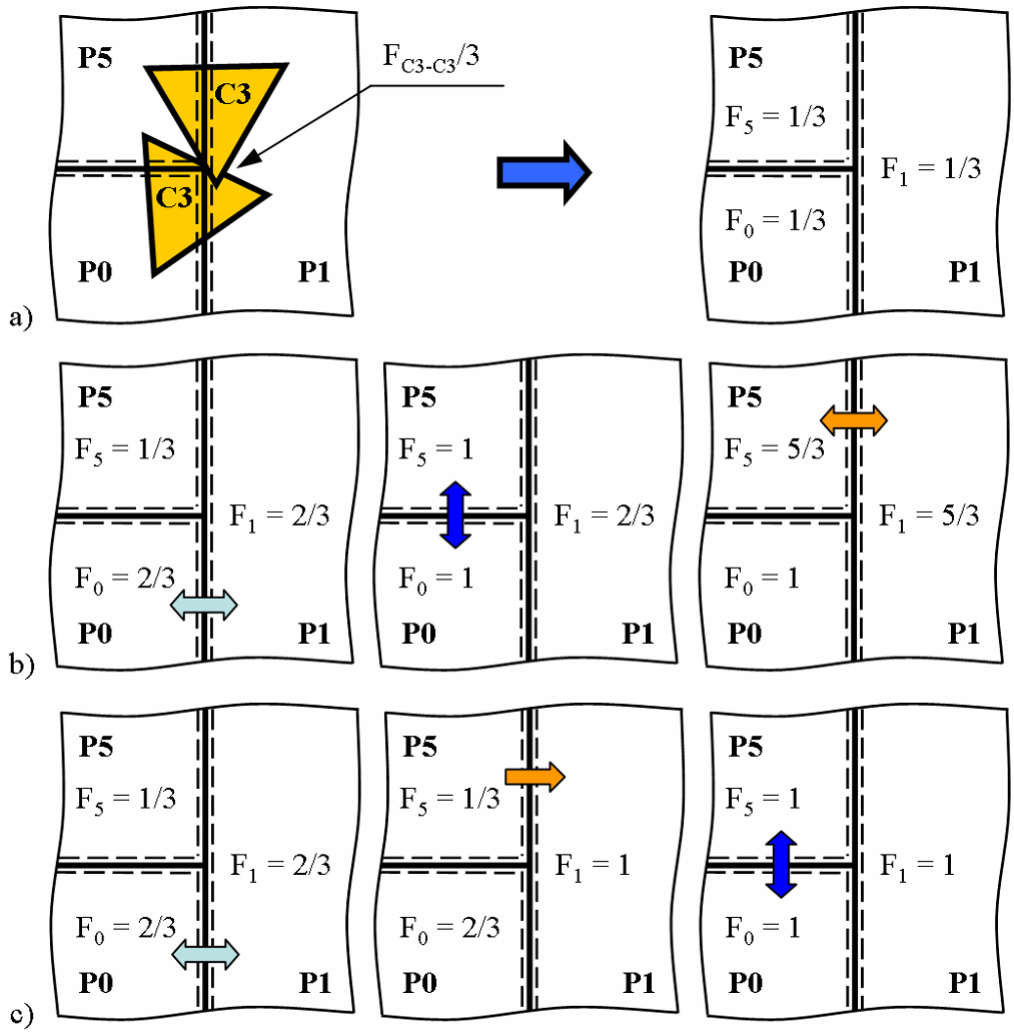


Figure 3.66: a) Contact force between two interfacial elements C3-C3 is equal to 1, b) Resulting contact force on each processor if messages are exchanged in random order, c) Resulting contact force on each processor if order of messages is set.

Setting an order of messages is particularly useful during exchange of nodal forces, for instance, contact force calculated from the interaction between two interfacial el-

ements C3-C3 located at the corner between processors 0, 1 and 5 (Figure 3.63) is divided by 3.0. Let us assume that the contact force is equal to 1, then contact force on each processor is equal to $1/3$, see Figure 3.66a. If the order of all messages was random, resulting contact force on each processor would be summed incorrectly (Figure 3.66b). In order to correctly add all forces on all processors, one of the horizontal messages must be sent in one direction only and before the vertical message. This can only be achieved if the order of messages is set in advance, see Figure 3.66c.

3.13 Parallel Input

Input/Output (I/O) is an important part of any FDEM simulation and both can be quite expensive in terms of CPU time. This is true, especially for writing an output since this must be done every couple of hundred of steps, unlike reading input file which is done only once. MPI provides functions to perform I/O operations in parallel from version 2, however, MPI I/O functions can be used only for unformatted binary files.[58]

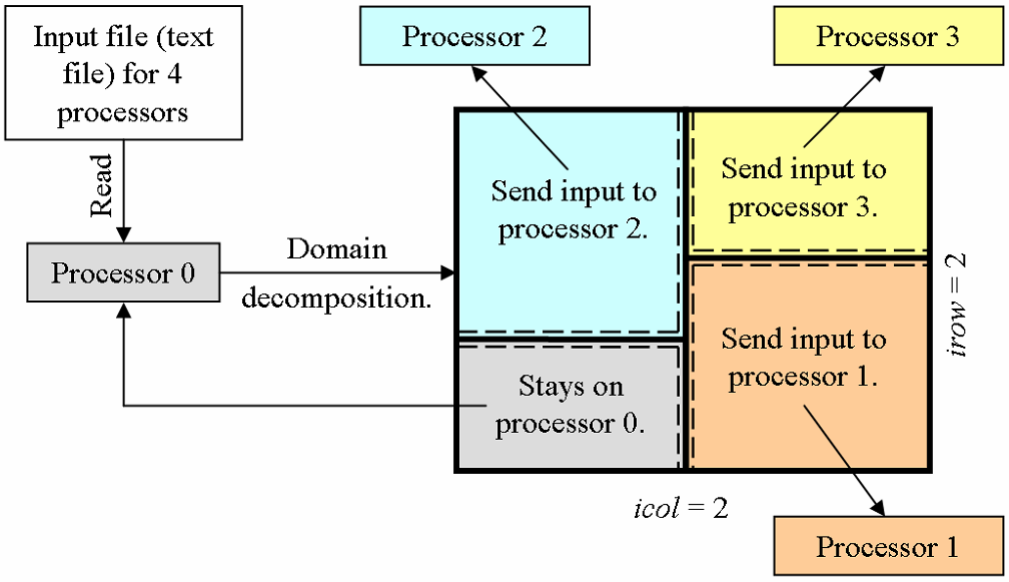


Figure 3.67: One processor (rank 0) reads input file, performs domain decomposition and distributes the data to remaining processors.

An input file in FDEM code is a text file and the formatting is quite complex. For this reason MPI I/O functions are not employed. Instead the input file is read by the processor with rank 0 which also performs domain decomposition (see Chapter 3.5) of the global computational domain. When the domain decomposition is finished,

processor 0 assigns sub-domains to the remaining processors and sends corresponding data to each processor, see Figure 3.67.

The disadvantage of this approach is that an increased CPU time is needed to perform input as well as increased RAM requirements on processor 0, since processor 0 has to read the whole input file and save to the memory. The amount of RAM memory will be, therefore, a limiting factor for large-scale simulations (hundreds of millions of discrete elements). If the input data is bigger than the amount of available RAM memory, it would lead to a memory swapping which would increase CPU time needed for input significantly.

3.14 Parallel Output

An output file in sequential FDEM code is a compressed text file. As mentioned above, MPI I/O functions give support for writing into an unformatted binary file, only the output file in the parallel implementation of the FDEM code is a binary file.

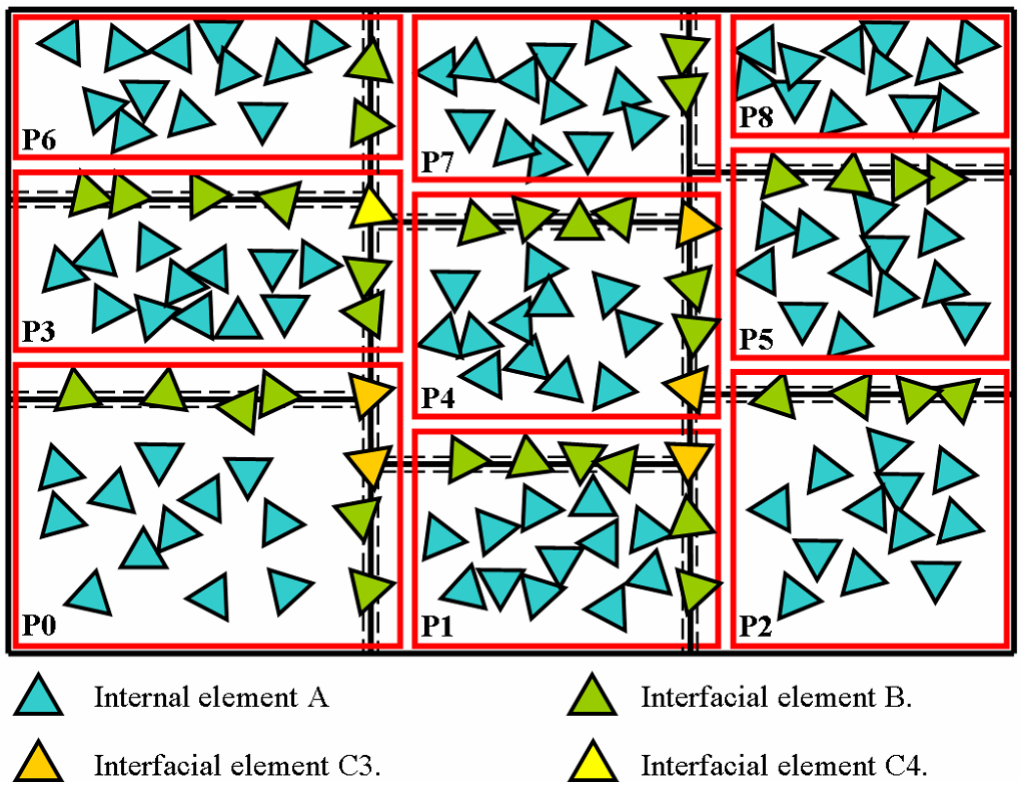


Figure 3.68: Output for each processor: internal elements A, interfacial elements at right and top borders B and C3 and interfacial elements C3/C4 at top-right corner.

Since interfacial elements are shared among two or more processors, it must be

ensured that no discrete/finite element is written into the output file twice. In order to do that each processor writes into output file:

- all internal elements A,
- all interfacial elements B located at right and top borders,
- all interfacial elements C3 located at right border,
- all interfacial elements C3 and C4 located at top-right corner, see Figure 3.68.

Each status and border/corner is assigned a unique flag (see Chapter 3.7), therefore, before the element is written into the output file, its flag is checked.

Output from each processor is written in parallel into a single output file which is shared between all processors by using MPI I/O functions. In order to write into the output file, one triangular or joint element, 3 integer numbers and 12 floating point numbers are needed, therefore, a structure **YPOD** is created. All elements which must be written into the output on each processor are first copied into an array of the structure **ypod** and a count of elements $elsum_i$ (i-th processor) is recorded. These counts from each processor are gathered by using MPI_Allgather() function.

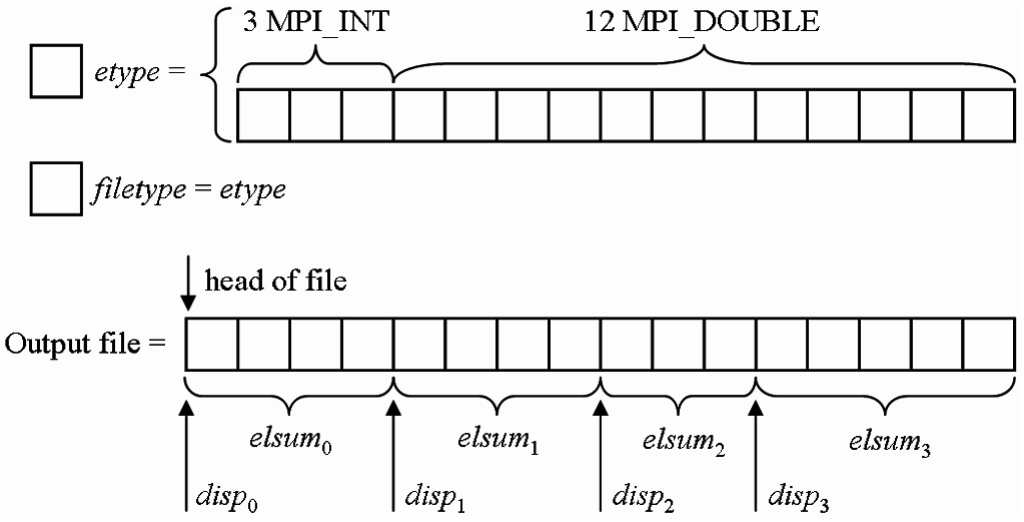


Figure 3.69: A shared output file for four processors Numbers of elements $elsum_i$ on processors 0, 1, 2 and 3 are equal to 4, 4, 3 and 5, respectively.

Algorithm 3.10 Parallel output written into a shared output file.

```
1:  static integer iffirst = 0;                          ▷ If first output.
2:  YPOD *ypod;                          ▷ Array of structure YPOD for output.
3:  integer elsum, isum;                      ▷ Count of elements - local/global.
4:  MPI_Aint extent;                                      ▷ Extent for etype.
5:  MPI_Offset disp;                          ▷ Position from which to write.
6:  MPI_File fout;                                        ▷ Output file.
7:
8:  Allocate ypod;                          ▷ Triangular plus joint elements.
9:  elsum = 0;
10: for (i = 0; i < nelem; i++) do                        ▷ Triangular elements.
11:     if (flag of element == output) then
12:         Save output for element into ypod;
13:         elsum = elsum + 1;
14:     end if
15: end for
16: for (i = 0; i < njtem; i++) do                        ▷ Joint elements.
17:     if (flag of element == output) then
18:         Save output for element into ypod;
19:         elsum = elsum + 1;
20:     end if
21: end for
22: MPI_Allgather(elsum);                  ▷ Gather all counts on all processors.
23: isum = 0;
24: for (i = 0; i < rank; i++) do                  ▷ Rank of current processor.
25:     isum = isum + elsum_i;                ▷ Sum counts until i = rank − 1.
26: end for
27: MPI_File_open(name, &fout);                ▷ Open file and check for error.
28: if (iffirst == 0) then                                ▷ Only first output.
29:     YodBuildTypeMem(&tyodm, ypod);          ▷ Build datatype for writing.
30:     YodBuildEtype(&tyode, fout);                ▷ Build datatype for etype.
31:     iffirst = 1;
32: end if
33: MPI_File_get_type_extent(fout, tyode, &extent);          ▷ Get extent for etype.
34: disp = isum · extent;                     ▷ Position from which to write.
35: MPI_File_set_view(fout, disp, tyode, tyode, "native", MPI_INFO_NULL);
36: MPI_File_write(fout, &ypod[0].i1int[0], elsum, tyodm, MPI_STATUS_IGNORE);
    ▷ Start writing whole ypod to the file from set file view.
37: MPI_File_close(fout);
38: free(ypod);
```

A so-called *etype* is created from the structure (Figure 3.69) by using functions for building a derived datatype and its *extent* is found. This is analogical to sending

a message but, instead of copying the structure into the message, it is written into the output file. Counts of elements from each processor, together with an *extent* of the *etype*, give each processor enough information to calculate the position in shared output file $disp_i$ from which to start writing output, see Figure 3.69. The content of the whole array is then written into the output. The whole process is summarised in Algorithm 3.10.

Chapter 4

VERIFICATION AND PERFORMANCE TESTS OF THE DEVELOPED PARALLEL SOLUTIONS

4.1 Introduction

As the main aim of the parallelisation is an increase of the computational power, it is necessary to test and verify the parallel implementation of the FDEM code and assess its performance. Ideally the verification study would be carried out by comparing results obtained by the parallel code for different number of processors with results obtained by the sequential code. This is not possible due to the presence of rounding errors which cause differences in the position of discrete elements. Thus, a general trend in the motion of all discrete elements during the run of the simulation combined with a comparison of total kinetic energies (sum of kinetic energy of all particles) obtained for a different number of processors is used for a verification purpose. Further verification and performance tests are provided in Chapter 5.

All examples in this thesis were run on a HPC cluster with 3592 nodes. Each node contains two 8-core 2.60 GHz Intel Xeon E5-2670 CPUs. 2395 nodes have 32 GB DDR3 1600MHz of RAM memory each, 1125 nodes have 64 GB of RAM and 72 nodes have 128 GB of RAM memory. Therefore, the maximum memory available for one processor/core is 8 GB of RAM.

4.2 Numerical Example

To illustrate the load balancing and re-partitioning, a box filled by 32400 discrete elements, each comprising 6 finite elements (199496 finite elements in total including elements comprising the bounding box), with initial velocity $v = 100\,\mathrm{m/s}$, was tested on up to 48 processors, see Figure 4.1. The direction of the velocity is given by an angle α measured from horizontal direction. The angle for each particle was randomly generated in the range $0^\circ < \alpha < 90^\circ$. The acceleration of gravity is set to zero in order to enable particles to move diagonally across the box. Material properties of each discrete element are as follows: Modulus of elasticity E = 990 MPa, Poisson's ratio $\nu = 0.5$, density $\rho = 1000\,\mathrm{kg/m^3}$ and contact penalty is 1.32 GPa.

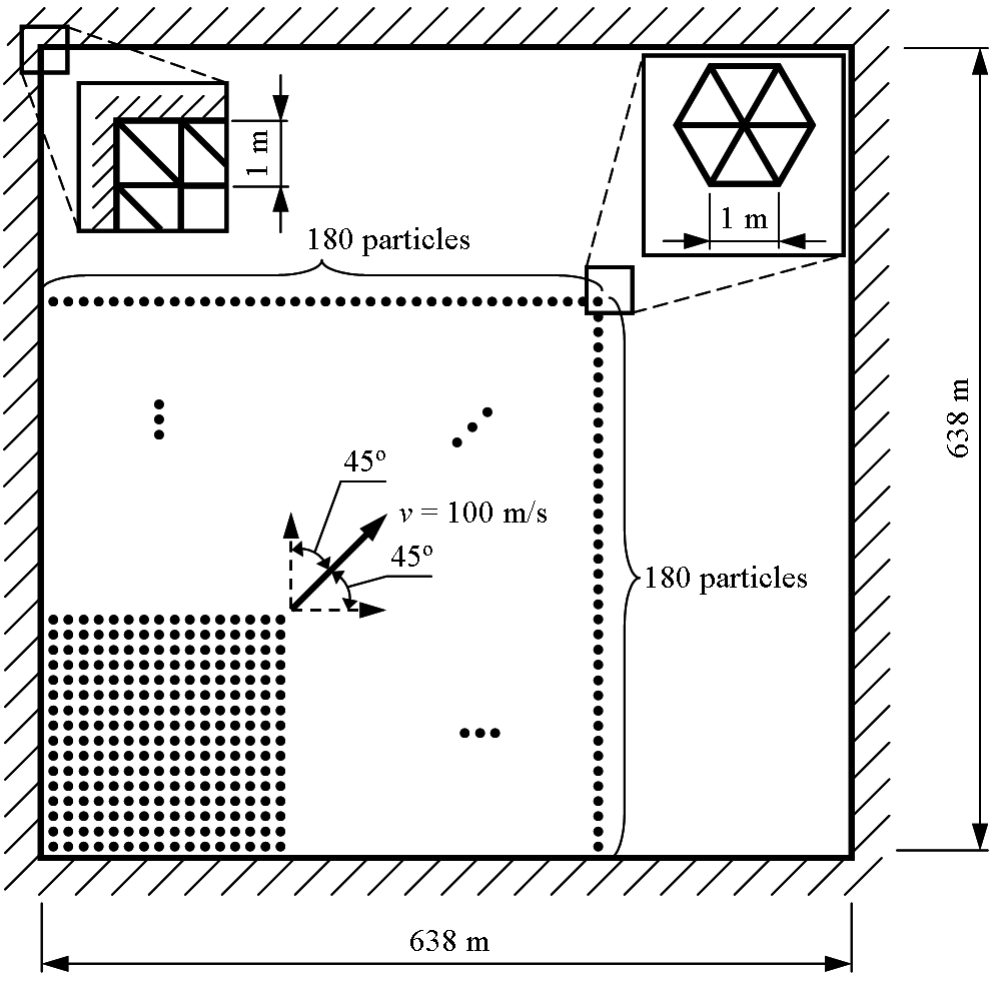


Figure 4.1: Initial configuration for a box filled with 32400 particles each comprising 6 finite elements. All particles are assigned initial velocity 100 m/s with a random direction given by an angle α.

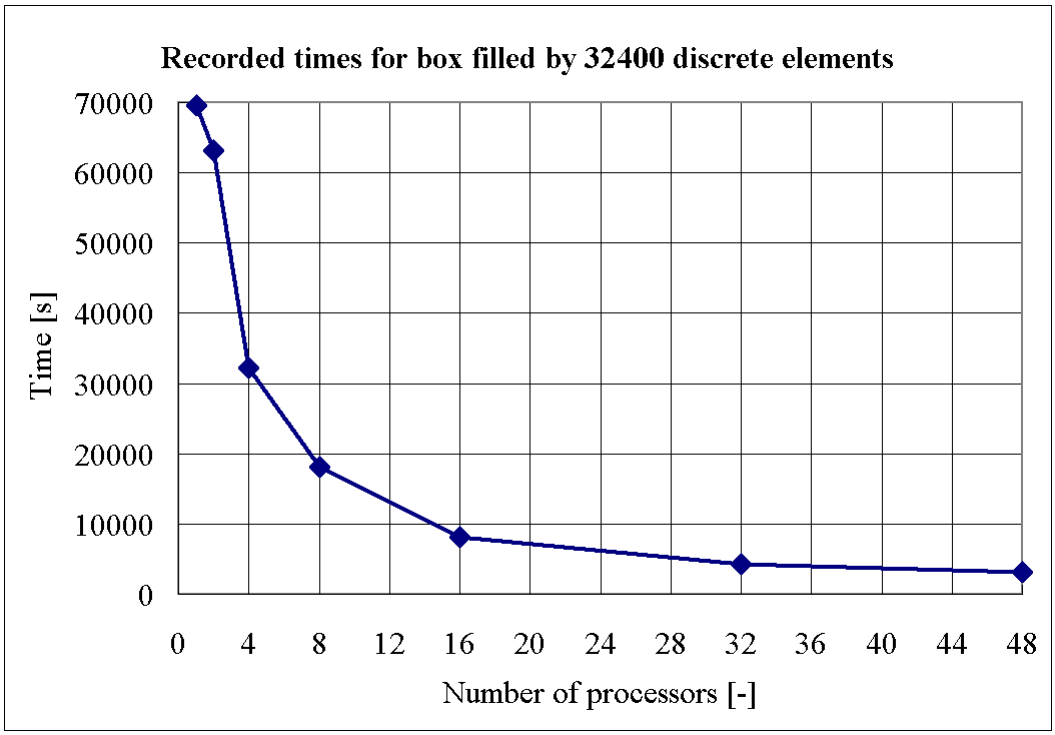


Figure 4.2: Recorded times for a box filled with 32400 particles.

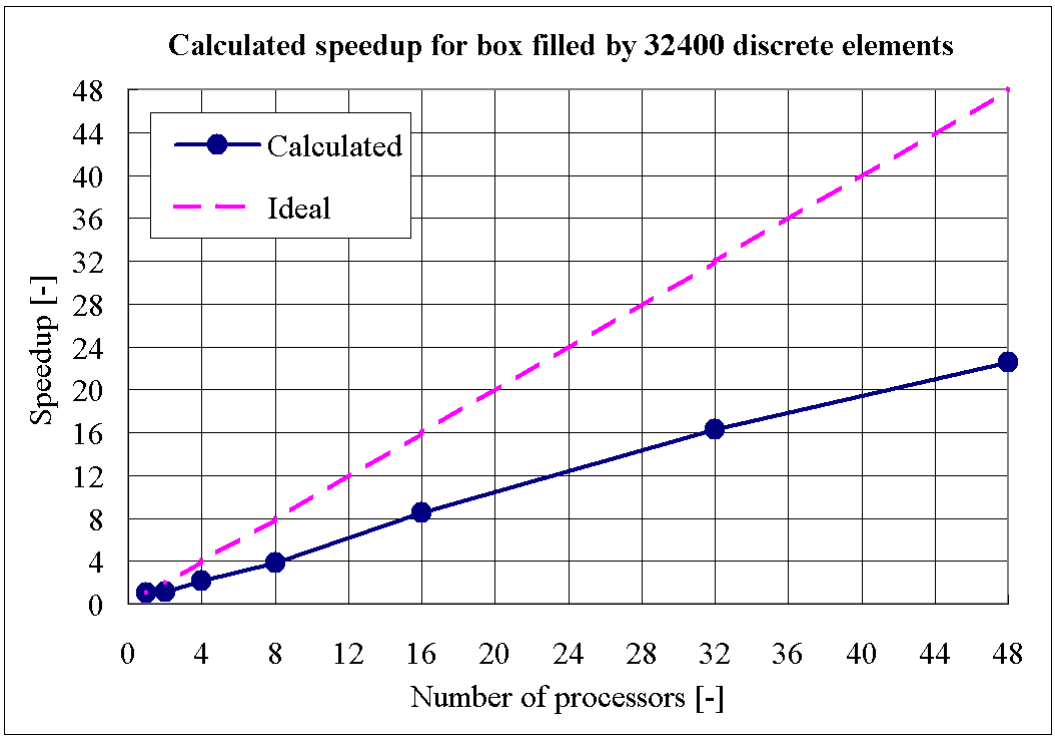


Figure 4.3: Calculated speedup for a box filled with 32400 particles.

Number of processors	CPU Time [s]	Speedup [-]
1	69526	1
2	63087	1.10
4	32179	2.16
8	18068	3.85
16	8127	8.55
32	4270	16.28
48	3087	22.52

Table 4.1: Recorded CPU times and calculated speedup for a box filled with 32400 particles.

Recorded simulation times for different numbers of processors and calculated speedup are summarized in Table 4.1. Simulation times and speedup for up to 48 processors are plotted in Figures 4.2 and 4.3.

The speedup obtained for the above numerical example can be considered a worst case scenario. Even though the initial velocity ranges from 0° to 90°, after some initial collisions of particles, the majority of particles move diagonally across the box. It follows, that for the bigger part of the simulation time, the CPU expensive contact interaction between particles is limited. Thus, the amount of computation is smaller in each time step than a typical FDEM simulation and the ratio between computation and parallelisation overhead is also smaller resulting in decreased performance.

Since the majority of particles move in a diagonal direction, the main part of parallelisation overhead consists of computation and communication during element migration and, also, redistribution of elements during load balancing. It can be seen from the results (Table 4.1) that the parallelisation overhead takes approximately half of the simulation time.

The speedup between 4 and 32 processors has a nearly linear trend (see Figure 4.3) and it is roughly equal to half of ideal speedup, see Table 4.1. The speedup obtained for 2 processors is very low due to the fact that the size of messages (number of elements located at the border between processors) is quite big. This is also true for speedup obtained for 8 processors, since the grid of processors have 2 rows and 4 columns thus the ratio between horizontal and vertical dimensions of the sub-domain is double the ratio for e.g. 4 or 16 processors (2×2 or 4×4). This corresponds with an increase in communication times. It can be observed from the results in Table 4.1 that the speed up for 4 and 16 processors is higher than half of the ideal speedup while 2 and 8 processors have speedup lower than half of the ideal speedup.

The performance improves up to 32 processors, see Table 4.1. The results suggest that the communication cost scales well with the increasing number of processors. The performance drops for 48 processors due to the decreasing number of elements assigned to each sub-domain (roughly 4200 elements). The number of elements for 32 processors is around 6300 thus for the above numerical example the point where the parallelisation becomes too costly is around 5000 elements. This cannot be tested due to the cluster rules: the number of processors must be in multiples of 16 (if higher than 16 processors).

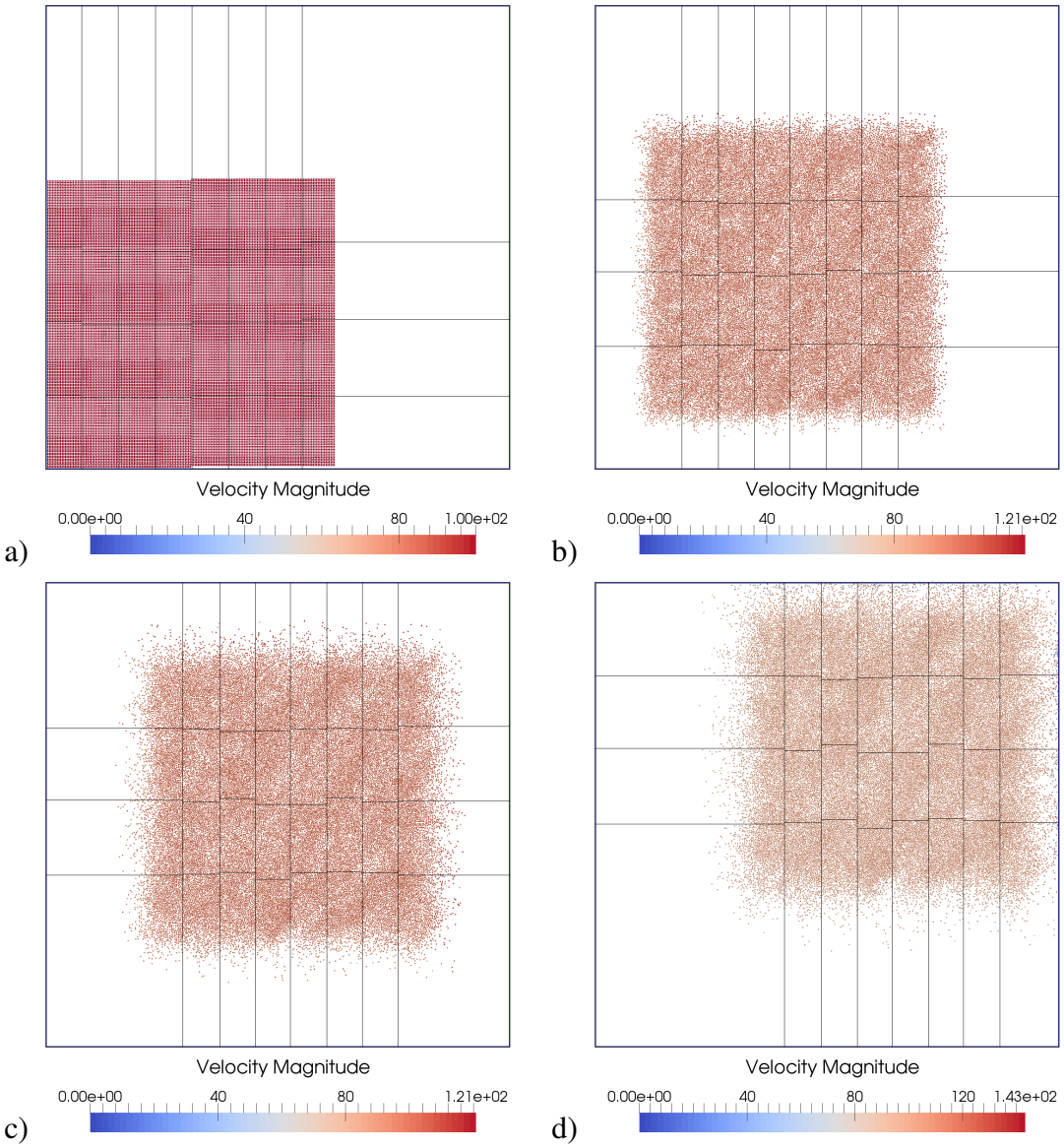


Figure 4.4: A motion sequence for a box filled with 32400 particles executed on 32 processors. a) Time 0 s. b) Time 1 s. c) Time 2 s. d) Time 3 s. The velocity is in m/s.

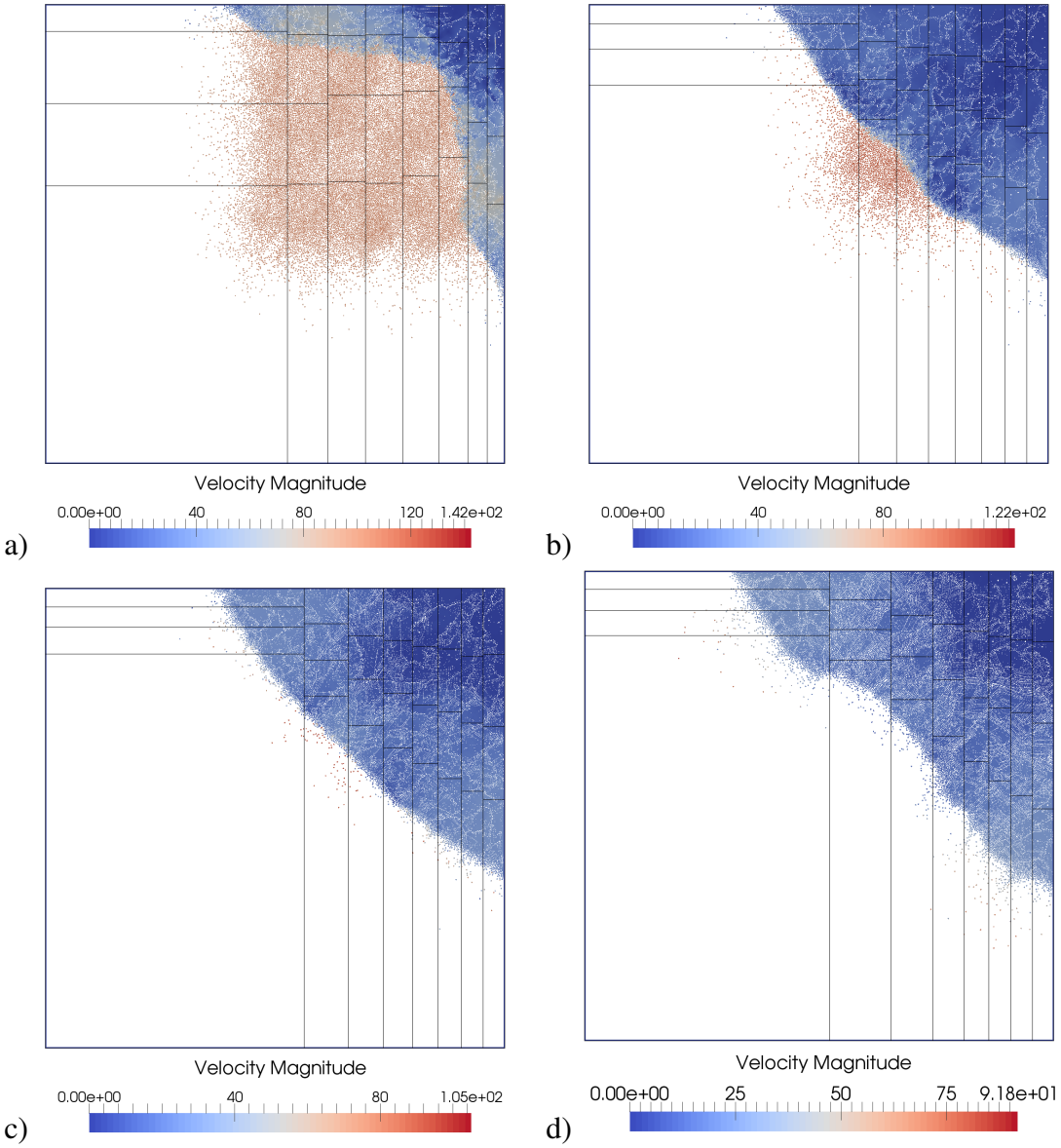


Figure 4.5: A motion sequence for a box filled with 32400 particles executed on 32 processors. a) Time 4 s, b) Time 5 s. c) Time 6 s. d) Time 7.5 s. The velocity is in m/s.

A motion sequence for the box filled by 32400 particles, executed on 32 processors at eight different times is shown in Figures 4.4 and 4.5. Domain decomposition for 2, 4, 8, 16, 32 and 48 processors is shown in Figures 4.6 and 4.7. It can be observed that the sizes of sub-domains in domain decomposition vary greatly, in order to achieve

balanced workload, and the ratio between horizontal and vertical dimensions of the sub-domain is quite high, especially for higher numbers of processors.

It should be noted that all the results for the above example are obtained for the following settings: the size of the buffer zone as specified in Chapter 3.5 is equal to the size of contact detection buffer and the maximum imbalance is set to 20 % since the problem is a highly dynamical one. Thus, if the maximum imbalance is set lower, the frequency of load balancing would be higher which is not desirable, since load balancing is an expensive operation.

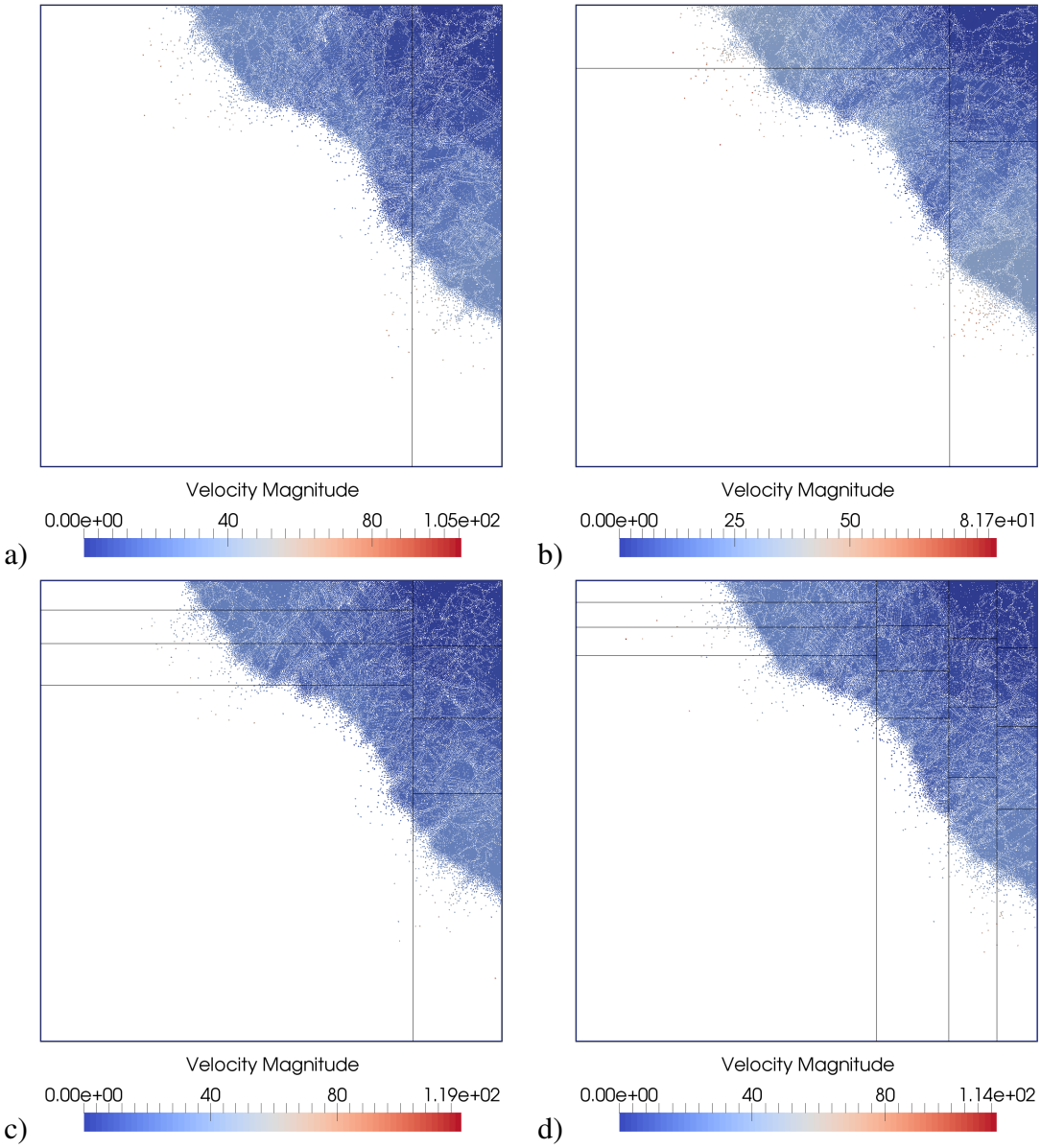


Figure 4.6: Domain decomposition for a box filled with 32400 particles at time 7.5 s on a) 2, b) 4, c) 8 and d) 16 processors. The velocity is in m/s.

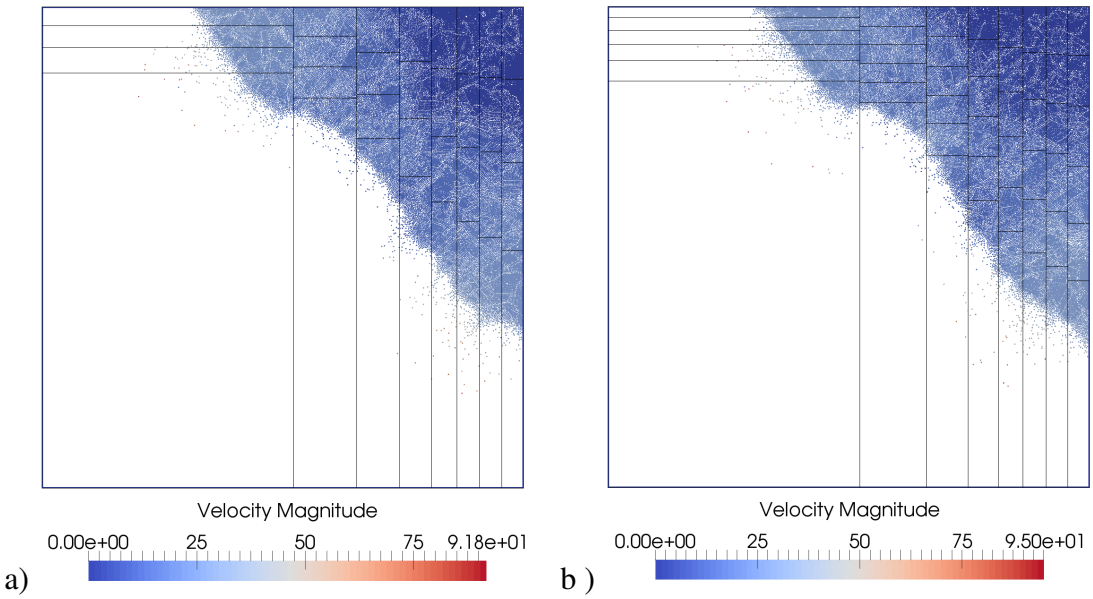


Figure 4.7: Domain decomposition for a box filled with 32400 particles at time 7.5 s on a) 32 and b) 48 processors. The velocity is in m/s.

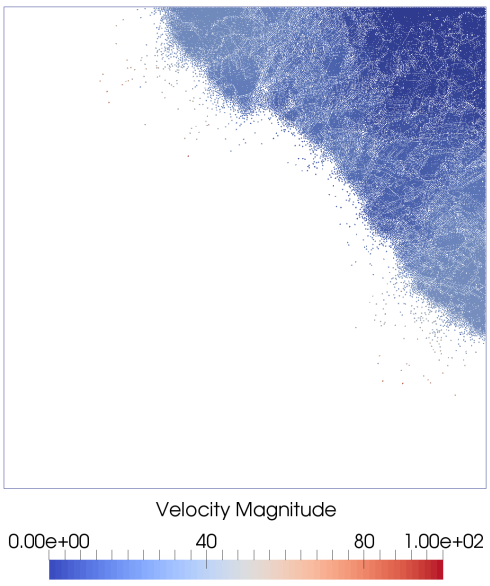


Figure 4.8: The box filled with 32400 particles at time 7.5 s executed on 1 processor. The velocity is in m/s

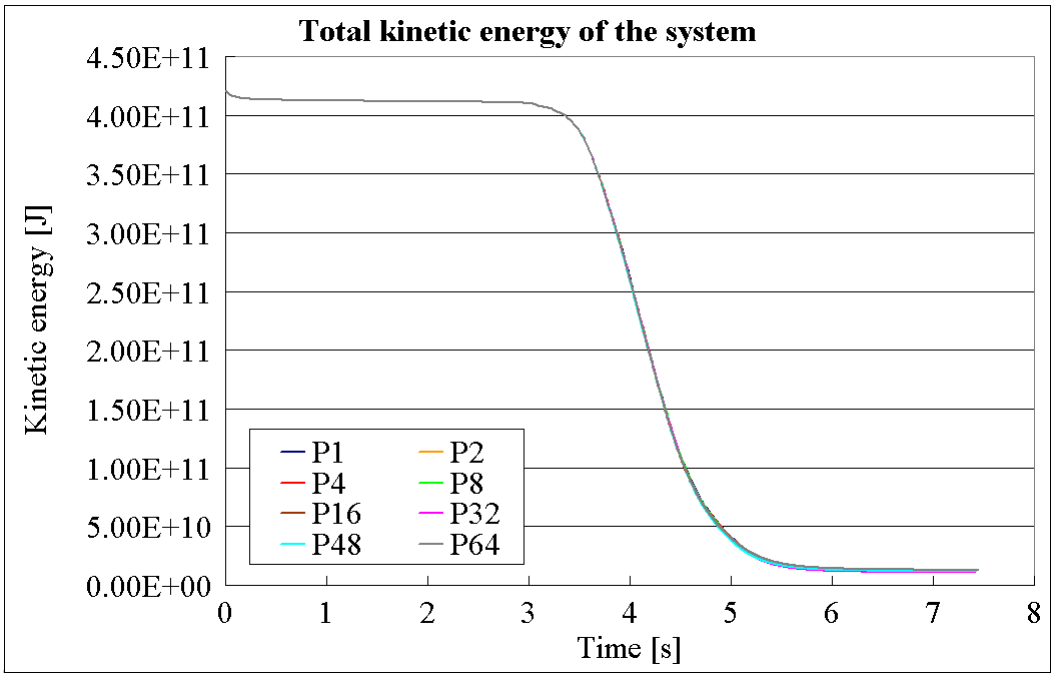


Figure 4.9: Total kinetic energy of the system for the box filled with 32400 particles for the whole simulation time.

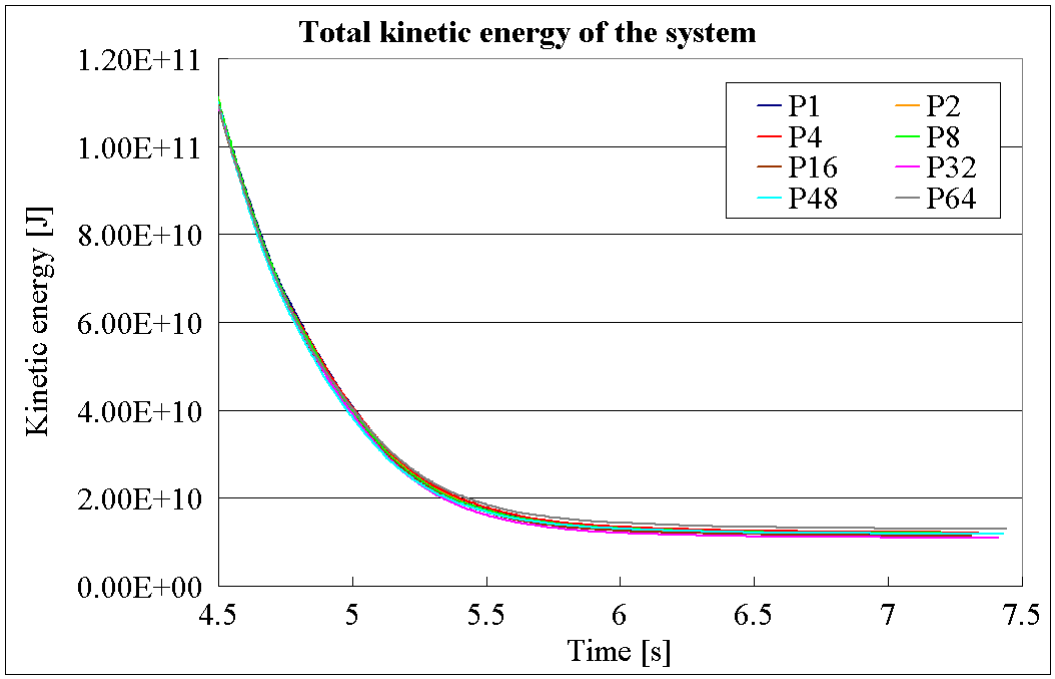


Figure 4.10: Total kinetic energy of the system for the box filled with 32400 particles from 4.5 s to 7.5 s.

The results obtained for different numbers of processors (Figures 4.6and 4.7) com-

pared with results obtained from a sequential code (Figure 4.8) show a good agreement. The general trend in the motion of particles is preserved in all simulations. The comparison of total kinetic energy of the system obtained for different numbers of processors shows a very good agreement, see Figure 4.9. The first discrepancy in the kinetic energy occurs around the time 4.5 s (see Figure 4.10) but even at the end of the simulation time the difference between kinetic energies is quite small.

4.3 Conclusions

The performance of the parallel implementation of proposed parallelisation solutions for FDEM compared with the performance of a sequential version of the code is good considering the highly dynamical nature of the test case. Further performance tests are carried out in Chapter 5.

The comparison of results (motion of particles and total kinetic energy of the system) shows a good agreement, thus confirming the validity of the developed parallelisation solutions.

Chapter 5

SOME APPLICATIONS OF THE DEVELOPED PARALLEL SOLUTIONS

5.1 Brazilian Disc Test

5.1.1 Introduction

The capabilities of the Combined Finite-Discrete Element Method are especially useful for simulating various problems in rock mechanics. Thus, all the applications in this chapter are chosen from this area of interest.

The Brazilian disc test is a common laboratory test originally designed by Berenbaum and Brodie[10] for indirect measurement of the tensile strength of brittle materials (rocks, concrete). The sample has a cylindrical shape with a diameter D and thickness t and it is loaded with an increasing force P at different rates of increase. The tensile strength σ is given by:

$$\sigma = \frac{2P_{max}}{\pi D t} \tag{5.1}$$

where P_{max} is the load applied at the time of the failure.

5.1.2 Definition of the Problem

The Brazilian disc test of a heterogeneous rock sample is numerically simulated on up to 64 processors. The radius of the disc is 38 mm and the disc has a unit thickness. The disc comprises 52308 triangular elements and 75466 joint elements. The tested material is Barre granite. It is a heterogeneous rock consisting of approximately 24%

quartz, 68% feldspar and 8% biotite.[129] Material properties for the rock sample and loading platens are summarized in Table 5.1.[107] The loading velocity of platens is 0.5 m/s, see Figure 5.1. The input file was created by a graphical user interface for the Y2D code developed by Mahabadi.[108] The random distribution of minerals in the tested rock sample is shown in Figure 5.1 together with the dimensions of the disc and loading platens.

Parameter	Quartz	Feldspar	Biotite	Joint	Platens
Young's modulus (GPa)	80	70	20	-	191
Poisson's ratio (-)	0.17	0.29	0.2	-	0.29
Density (kg/m^3)	2600	2600	2800	-	8030
Shear strength (MPa)	3.15e10	3.15e10	3.15e10	3.15e10	-
Tensile strength (MPa)	6	6	4	5	-
Fracture energy (N/m)	50	50	50	50	-
Contact penalty (GPa)	80	70	20	-	1.91e4

Table 5.1: Material properties for the Barre granite Brazilian Disc test.

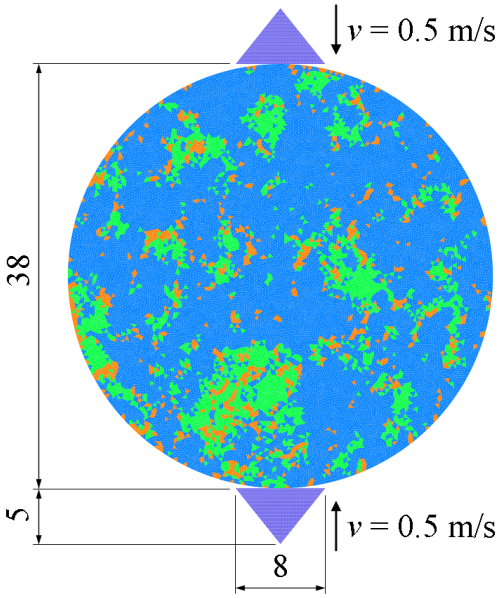


Figure 5.1: Material distribution and dimensions in mm for the rock sample. Green represents quartz, blue represents feldspar and orange represents biotite.

5.1.3 Results and Discussion

Recorded simulation times for different numbers of processors and calculated speedup are summarised in Table 5.2. Simulation times and speedup for up to 64 processors are plotted in Figures 5.2 and 5.3. It can be observed from the Figure 5.3 that the speedup has a super-linear trend on up to 32 processors. For more than 32 processors, the speedup is smaller than ideal speedup, which is expected, since with an increasing number of processors, the number of elements within each sub-domain decreases, closing to the point where the cost of parallelisation is not negligible anymore. In other words, the ratio between computation and communication is getting smaller, with an increasing number of processors, resulting in a decrease in performance for higher numbers of processors.

The efficiency of the parallel implementation from 2 to 64 processors is plotted in Figure 5.4. Except for one anomalous reading at 4 processors, it illustrates the steady decrease in the performance as the ratio between computation and communication gets smaller with an increasing number of processors. The most likely explanation for the decreased efficiency at 4 processors is a higher initial imbalance. As explained in Chapter 3.11, the partitioner creates boundaries of the sub-domain only at the boundaries of the load balancing cells. Thus, a small initial imbalance is expected, and its size depends on the finite element mesh, distribution of discrete elements in the computational domain and also on the number of processors. The performance for 64 processors is still acceptable even though the sub-domain contains only around 800 triangular elements and 1200 joint elements. Rapid decrease in performance is expected for even higher numbers of processors, thus it would be impractical to run the simulation on more than 64 processors.

Number of processors	CPU Time [s]	Speedup [-]	Efficiency [%]
1	155316	1	100
2	67598	2.30	114.88
4	35527	4.37	109.29
8	17205	9.03	112.84
16	8900	17.45	109.07
32	4763	32.61	101.90
48	3351	46.35	96.56
64	2652	58.57	91.51

Table 5.2: Recorded CPU times and calculated speedup and efficiency for the Brazilian Disc test.

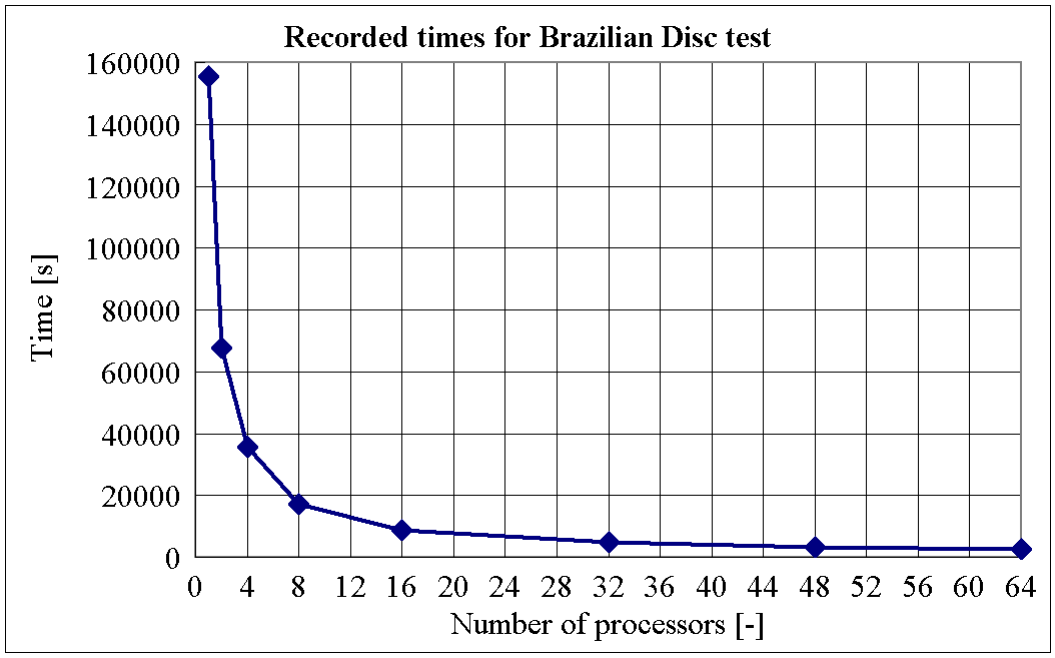


Figure 5.2: Recorded CPU times for Brazilian Disc test.

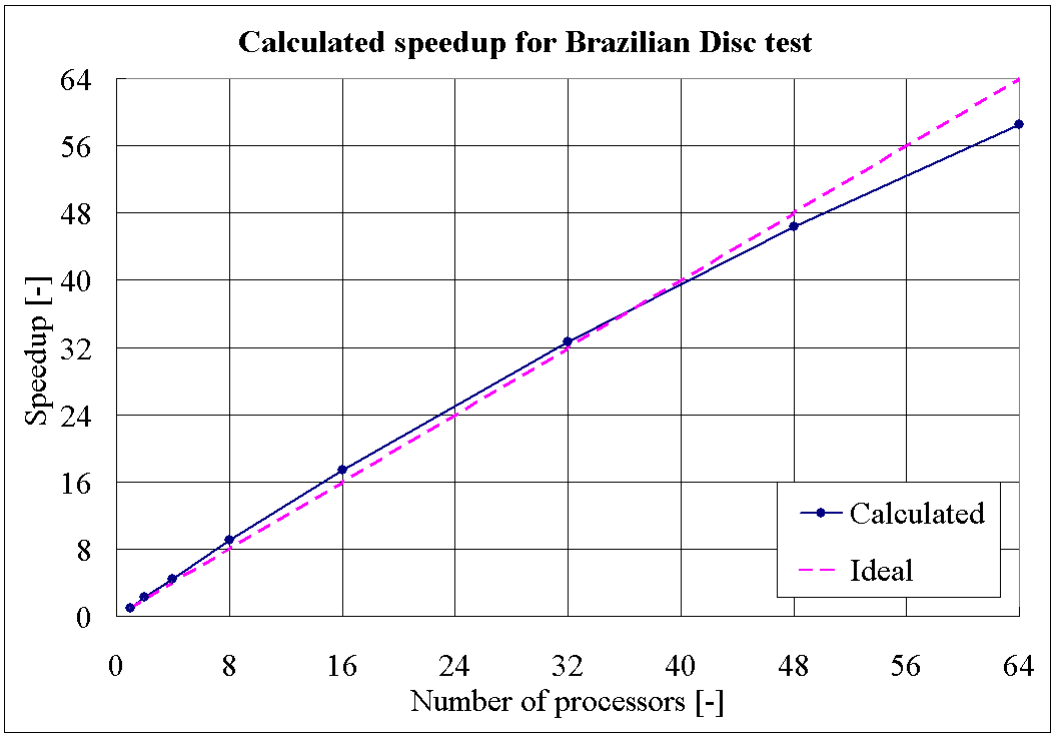


Figure 5.3: Calculated speedup for Brazilian Disc test.

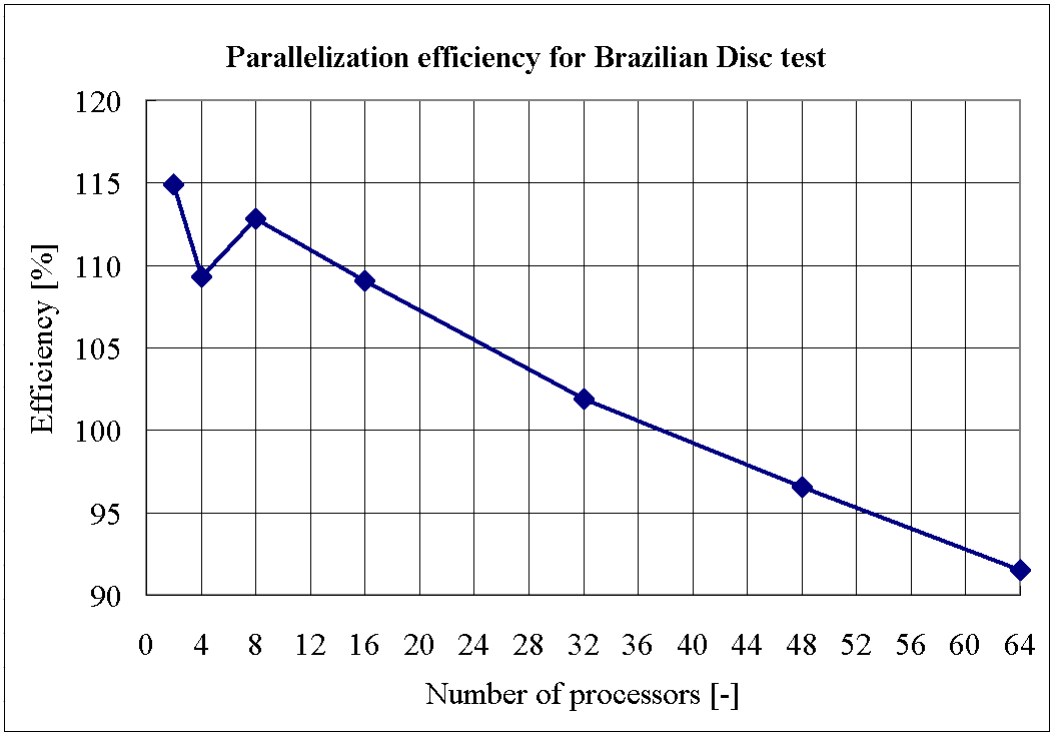


Figure 5.4: Calculated parallelisation efficiency for Brazilian Disc test.

The super-linear speedup for up to 32 processors can occur for a couple of different reasons. The first one is the increased amount of RAM compared with 1 processor. Another contributing reason can be the writing of output files. If different jobs at the cluster try to access the hard drive at the same time, it will result in competition for access. It has also been observed that the speedup is slightly lower, even though still super-linear, if the same problem is executed without recording the total kinetic energy of the system. This would suggest that the task of calculating and recording the total kinetic energy has a super-linear trend itself and, indeed, the performance of all test cases in this thesis improved if the kinetic energy was recorded.

A simulation sequence of the fracture pattern obtained for a Brazilian Disc test executed on 16 processors at six different times is shown in Figures 5.5 and 5.6.

Domain decomposition for 2, 4, 8, 16, 32 and 64 processors is shown in Figures 5.8 and 5.9. The fracture patterns in Figures 5.8 and 5.9 show a good correspondence with a fracture pattern obtained from a sequential code, see Figure 5.7. The comparison of the total kinetic energy of the system obtained for different numbers of processors shows a very good agreement up to approximately 0.07 s, see Figure 5.10. The first noticeable difference in the kinetic energy occurs around the time 0.03 s when first cracks appear, see Figure 5.11.

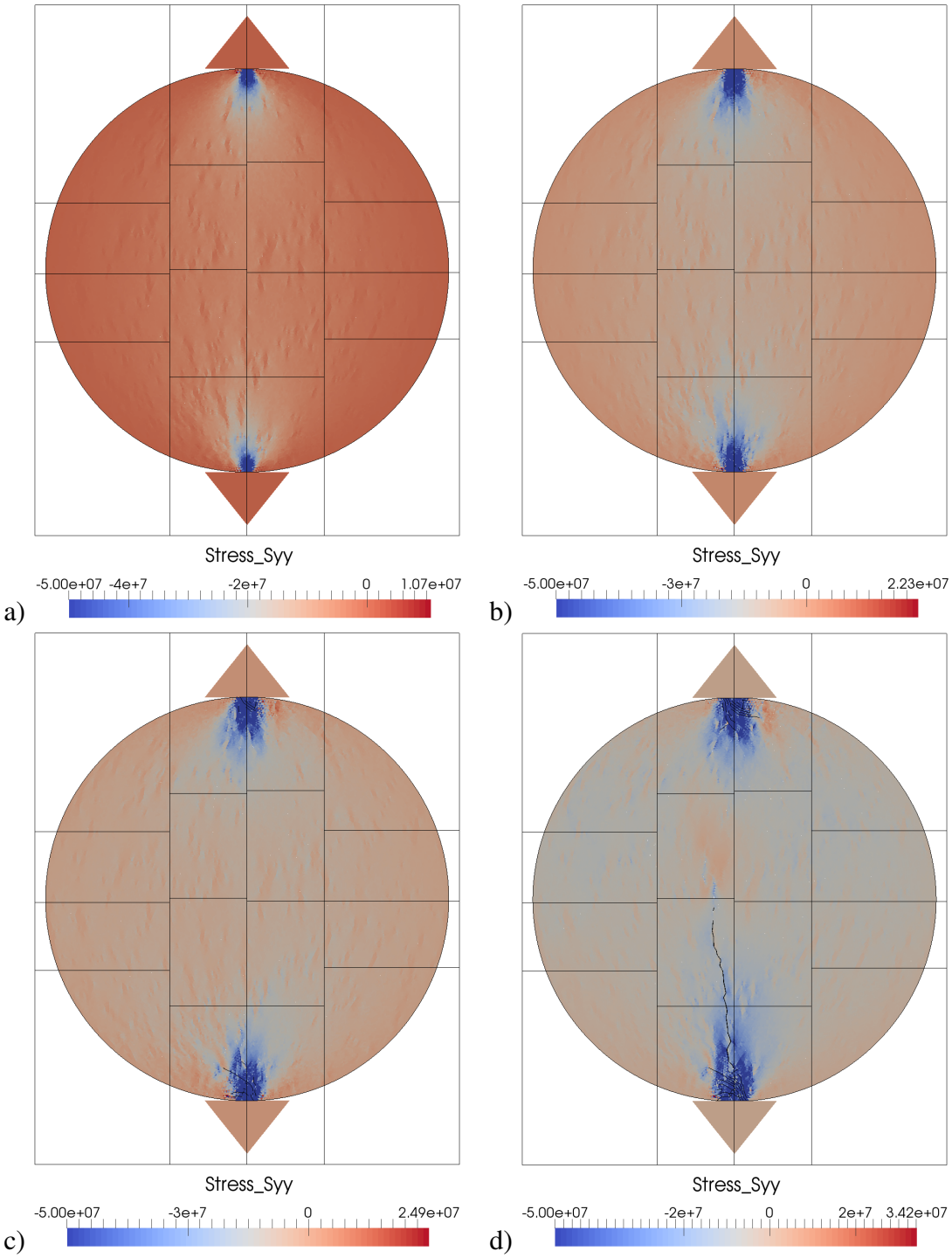


Figure 5.5: Brazilian Disc test for 16 processors at times: a) 0.025 s, b) 0.05 s, c) 0.075 s, d) 0.1 s. The stress σ_y is in Pa.

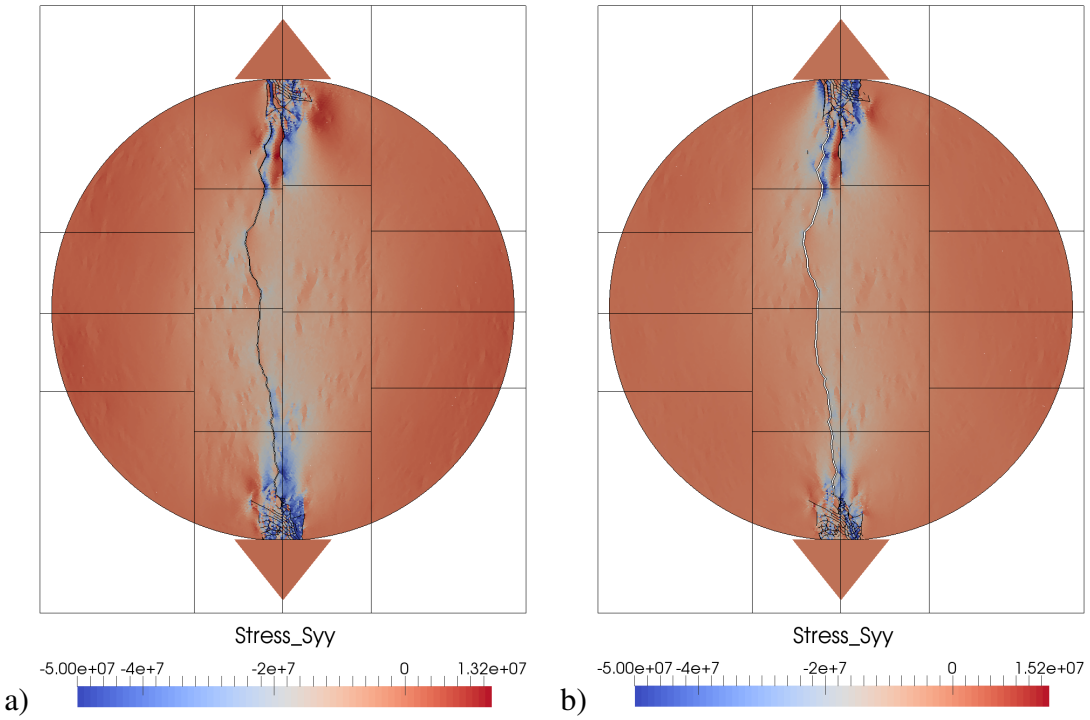


Figure 5.6: Brazilian Disc test for 16 processors at times: a) 0.125 s, b) 0.15 s. The stress σ_y is in Pa.

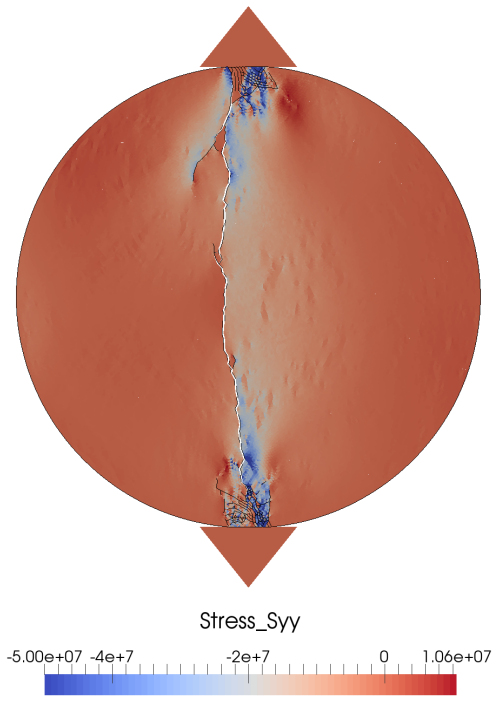


Figure 5.7: Brazilian Disc test at time 0.15 s on 1 processor. The stress σ_y is in Pa.

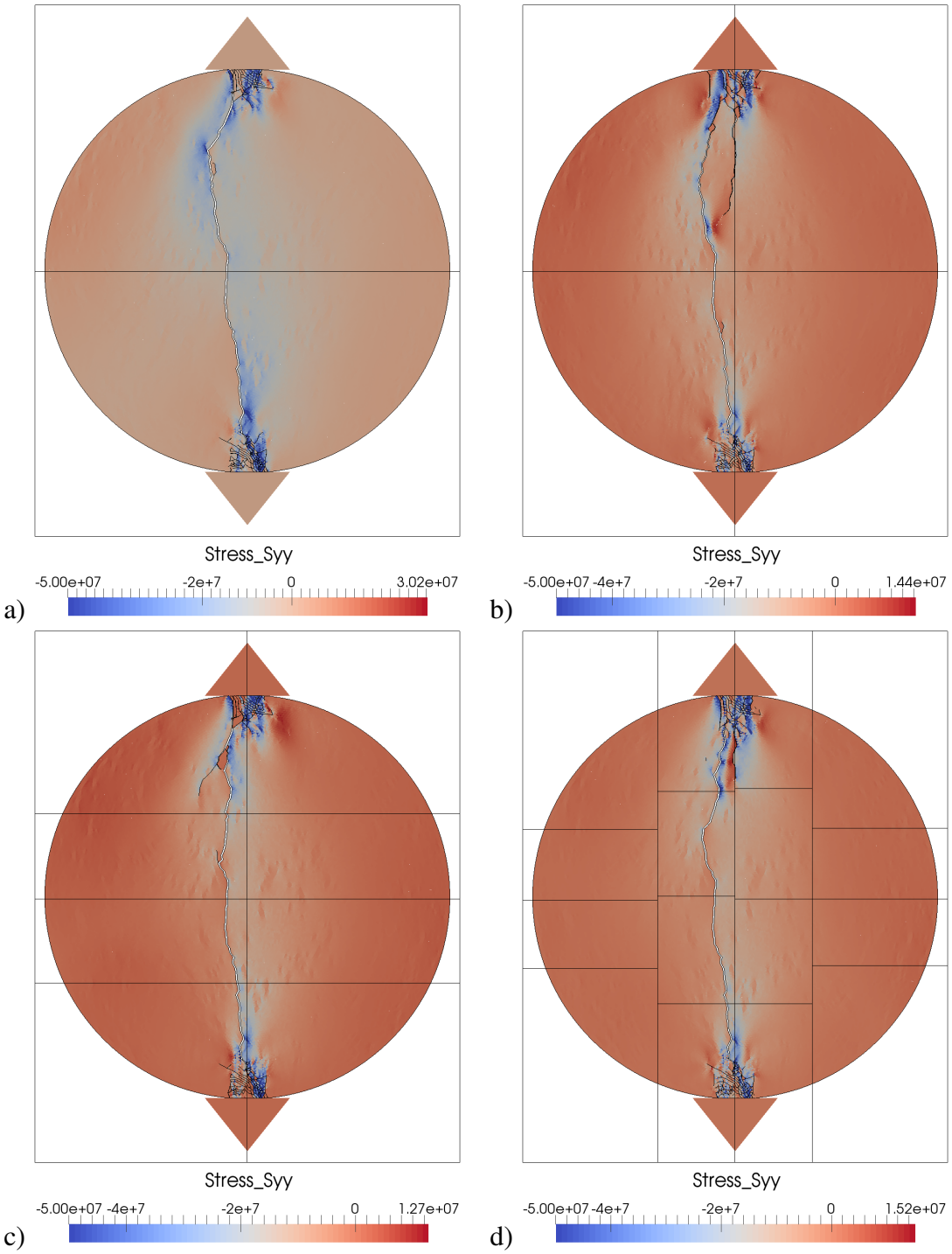


Figure 5.8: Domain decomposition for Brazilian Disc test at time 0.15 s on a) 2, b) 4, c) 8 and d) 16 processors. The stress σ_y is in Pa.

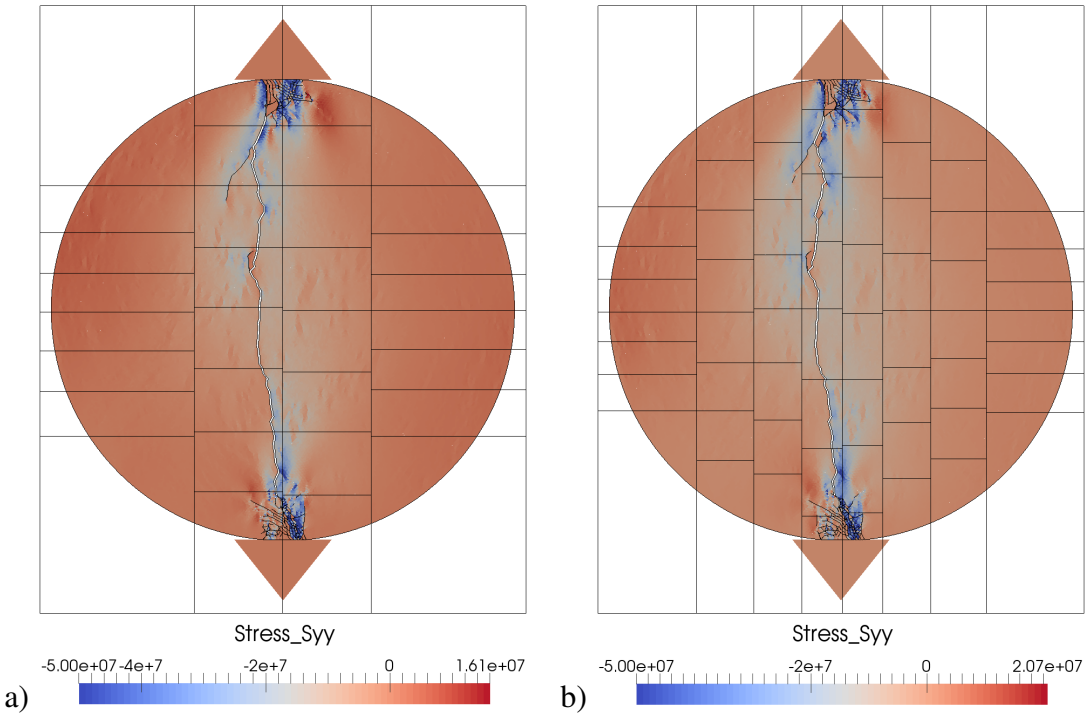


Figure 5.9: Domain decomposition for Brazilian Disc test at time 0.15 s on a) 32, b) 64 processors. The stress σ_y is in Pa.

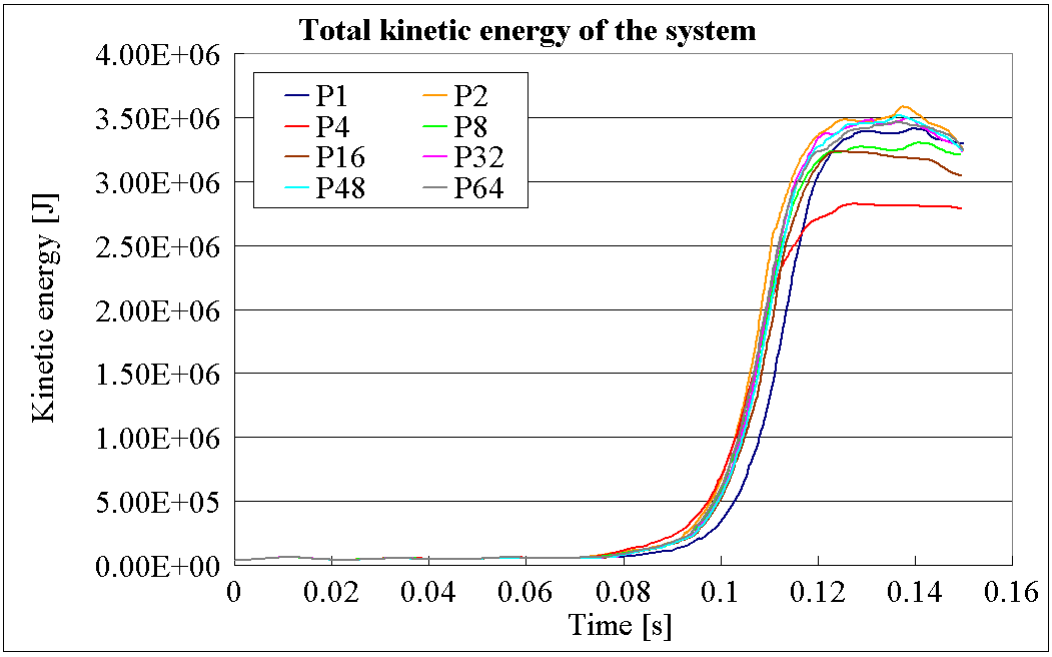


Figure 5.10: Total kinetic energy of the system for the Brazilian Disc test for the whole simulation time.

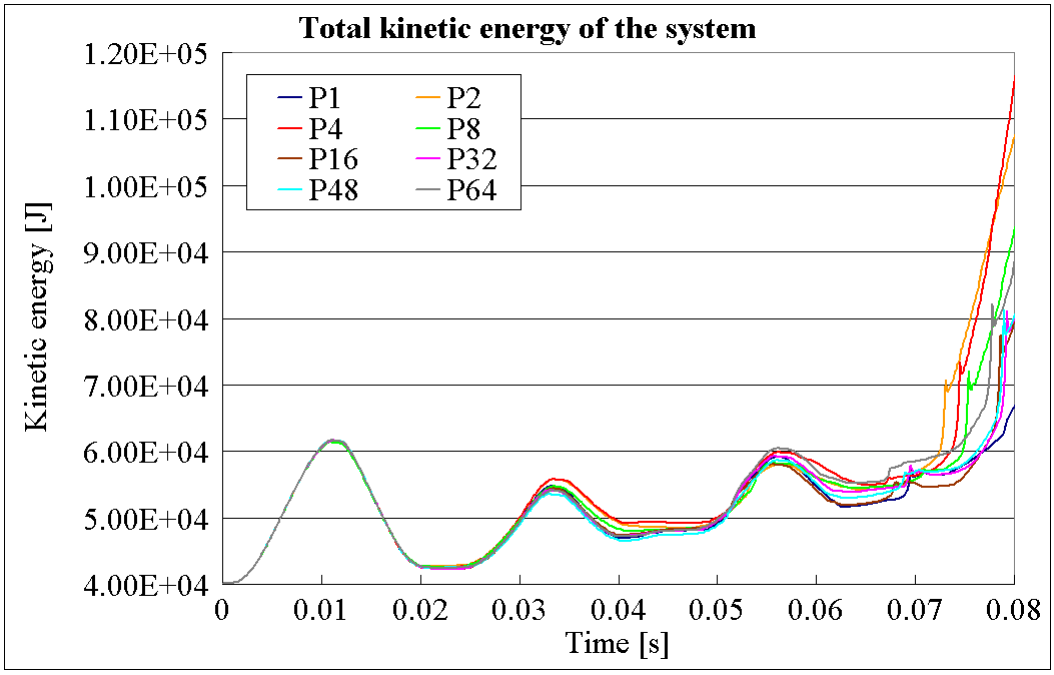


Figure 5.11: Total kinetic energy of the system for the Brazilian Disc test from 0 s to 0.08 s.

Understandably, the presence of rounding errors causes the difference in results obtained for the same input file which was run on different numbers of processors. This means only a general trend in the motion of discrete elements, or in this case a fracture pattern, must be assessed, since the fracture patterns for different numbers of processors evolve differently. A different fracture pattern (and different number of fractured/discrete elements) in each simulation is reflected by changes in the kinetic energy.

It should be noted that, even though the fracture patterns obtained are acceptable and corresponding with some previously published results,[108,107] the parallelised version of the FDEM code is based on the original version of Y2D code. Since then improvements have been made to deal with, for instance, quasi-static problems which also include Brazilian disc test simulation and a couple of different versions of the Y2D code now exist. A version of FDEM code named Y-Geo[110] would be better suited for the rock mechanics test cases presented in this chapter since Y-Geo addresses problems like; quasi-static friction, Mohr-Coulomb failure criterion, shear strength in rock joints, etc. The description of some of these improvements is presented by Mahabadi.[107]

5.2 Block Caving

5.2.1 Introduction

The block caving process offers an alternative to classic underground mining. The production rates achieved by the block caving can reach those achieved by surface mines. The rock, rich with ore, is undercut in the area below the rock. The rock, above the area which was undercut, may collapse due to the gravity if the rock is sufficiently massive and fractured. If the rock does not collapse naturally, then boreholes are drilled and the rock is blasted. Resulting blocks of rock fall down into prepared draw-bells.

Block caving is greatly influenced by the properties of the rock, geometry of the undercut area and other parameters. Methods of discontinua provide means to numerically investigate the influence of these parameters for the purpose of finding the optimum design of the block caving. FDEM is especially suitable for this kind of simulation due to its ability to handle transition from continua to discontinua as the block caving depends on the fracture and fragmentation processes. Thus the explosion in the borehole, fracture and fragmentation, as well as granular flow, can be all investigated together in one FDEM simulation.

The block caving triggered by an explosion in a borehole is, in essence, a FDEM coupled with Computational Fluid Dynamics (CFD) simulation. The implementation of the gas flow through a fracturing solid is based on a constant area duct compressible flow of ideal gas. It is out of the scope of this thesis to provide details on the theoretical background of this problem and its implementation. Theoretical background can be found in chapter 8 of FDEM book[117] which is based on a paper presented by Munjiza et al.[123]

5.2.2 Definition of the Problem

The rock blasting of a block of sandstone with dimensions 60×120 m is numerically simulated on up to 32 processors. Average values of the sandstone's material properties[98,50] were chosen. The propertis of triangular elements are: Modulus of elasticity E = 25.8 GPa, Poisson's ratio $\nu = 0.17$, friction coefficient $f_C = 0.5$, density $\rho = 2340\,\mathrm{kg/m^3}$ and contact penalty is 30 GPa. The properties of joint elements are: shear strength 3.78e7 GPa, tensile strength 3.15 MPa, fracture energy 250 N/m and fracture penalty 4 GPa. Shear strength is set to a high value in order to prevent fracturing in Mode II (fracture caused by shear stress).

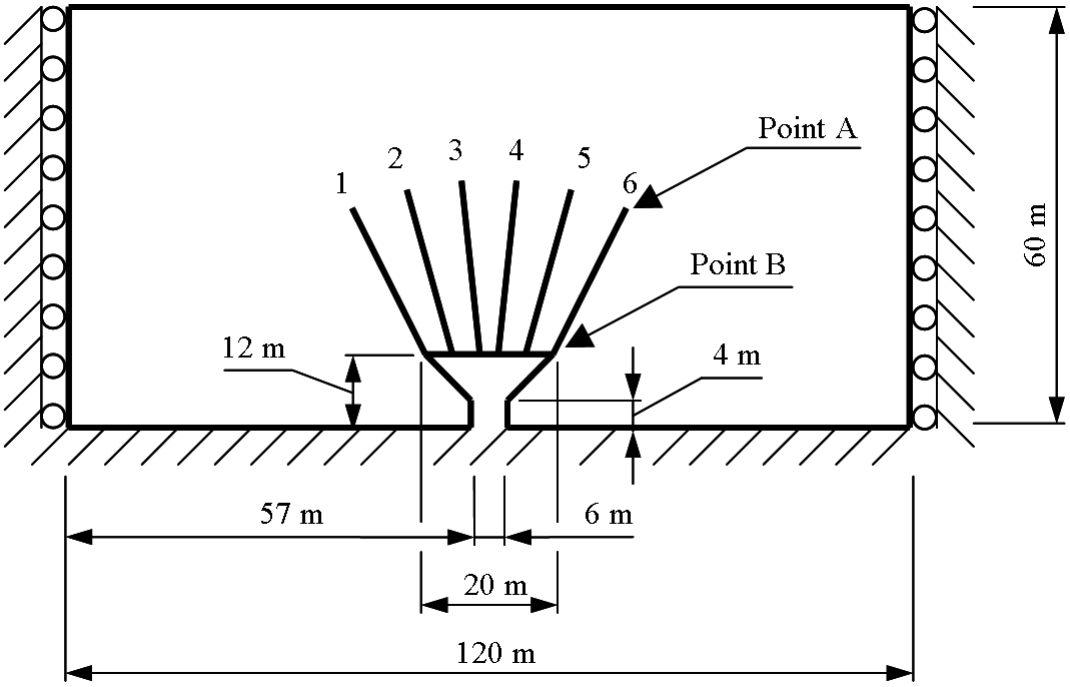


Figure 5.12: Block of rock with 6 boreholes.

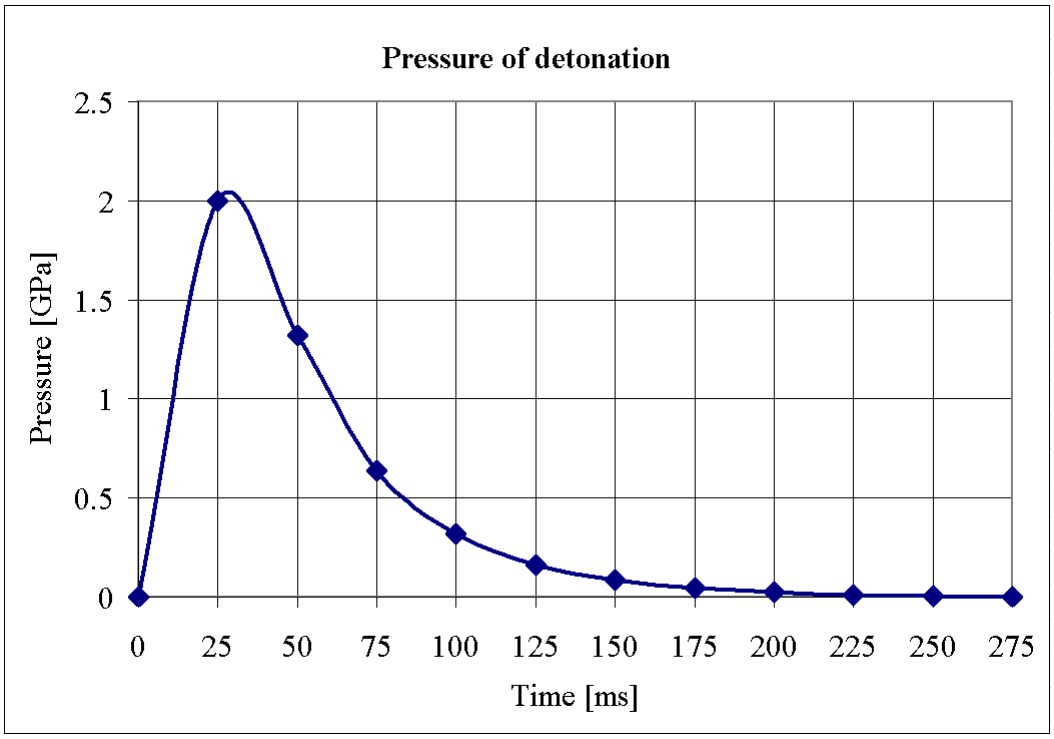


Figure 5.13: Amplitude of the pressure as a function of time.

Borehole	Point A		Point B		Explosion time	
	x [m]	y [m]	x [m]	y [m]	Start [s]	End [s]
1	-21.95	-27.22	-9.95	-48	3e-2	3.0275e-2
2	-14.18	-25.45	-5.97	-48	1.5e-2	1.5275e-2
3	-6.16	-24.36	-1.99	-48	0	2.75e-4
4	6.16	-24.36	1.99	-48	0	2.75e-4
5	14.18	-25.45	5.97	-48	1.5e-2	1.5275e-2
6	21.95	-27.22	9.95	-48	3e-2	3.0275e-2

Table 5.3: Coordinates of points A and B for each borehole together with start and end time of detonation at point A.

Vertical edges of the block are constrained in x direction and bottom horizontal edges are constrained in x and y direction, see Figure 5.12. The tip of each borehole is marked as point A and the other end is marked as point B. Coordinates of points A and B for each borehole are given in Table 5.3, together with start and end times of detonation at point A. Numbering of boreholes corresponds with numbering in Figure 5.12. The amplitude of the pressure in the borehole is 2 GPa and the pressure is a function of time, see Figure 5.13. Velocity of detonation is 5500 m/s. The mesh was generated with 0.7 m element size resulting in 32899 triangular elements and 50165 joint elements.

5.2.3 Results and Discussion

The block caving is numerically simulated on up to 32 processors. Recorded simulation times for different numbers of processors and calculated speedup and efficiency are summarised in Table 5.4. Simulation times, speedup and efficiency for up to 32 processors are plotted in Figures 5.14, 5.15 and 5.16.

The calculated speedup has a linear trend for up to 8 processors and after that the performance drops (Figure 5.15). This decrease of performance is also illustrated by the efficiency graph, see Figure 5.16.

Two main reasons exist for the decrease in performance. The first reason is the decreasing number of elements assigned to a sub-domain for higher numbers of processors, thus, the ratio between computation and communication gets smaller. For instance, for 32 processors each sub-domain has only around 1000 triangular elements and around 1500 joint elements.

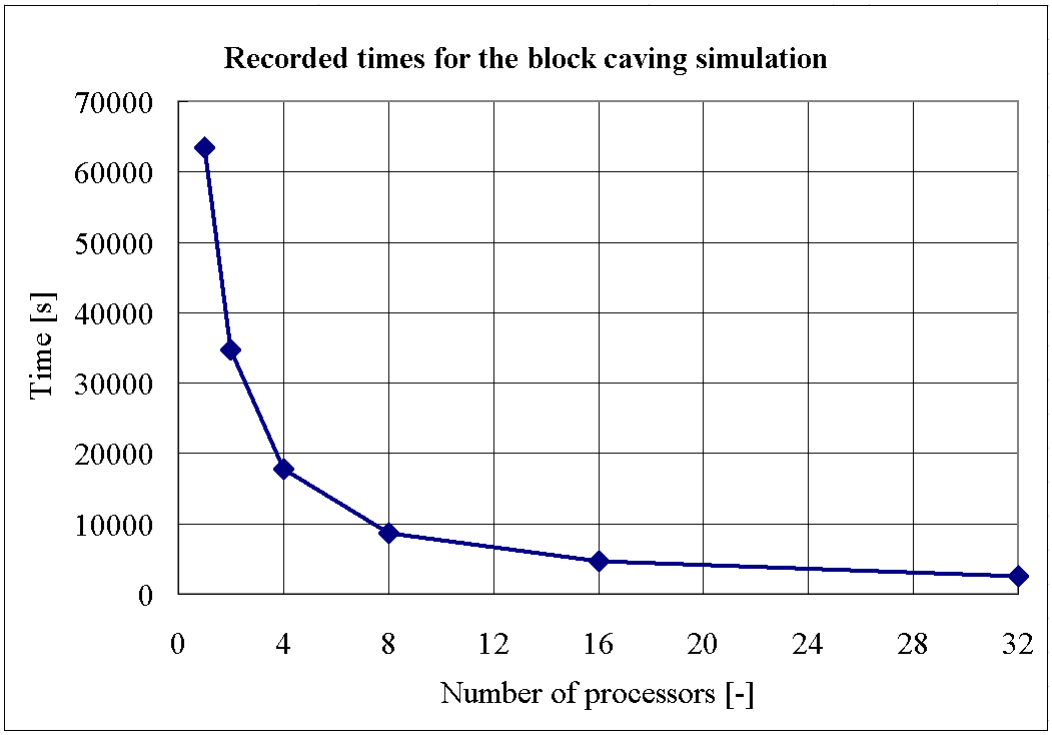


Figure 5.14: Recorded CPU times for the block caving simulation.

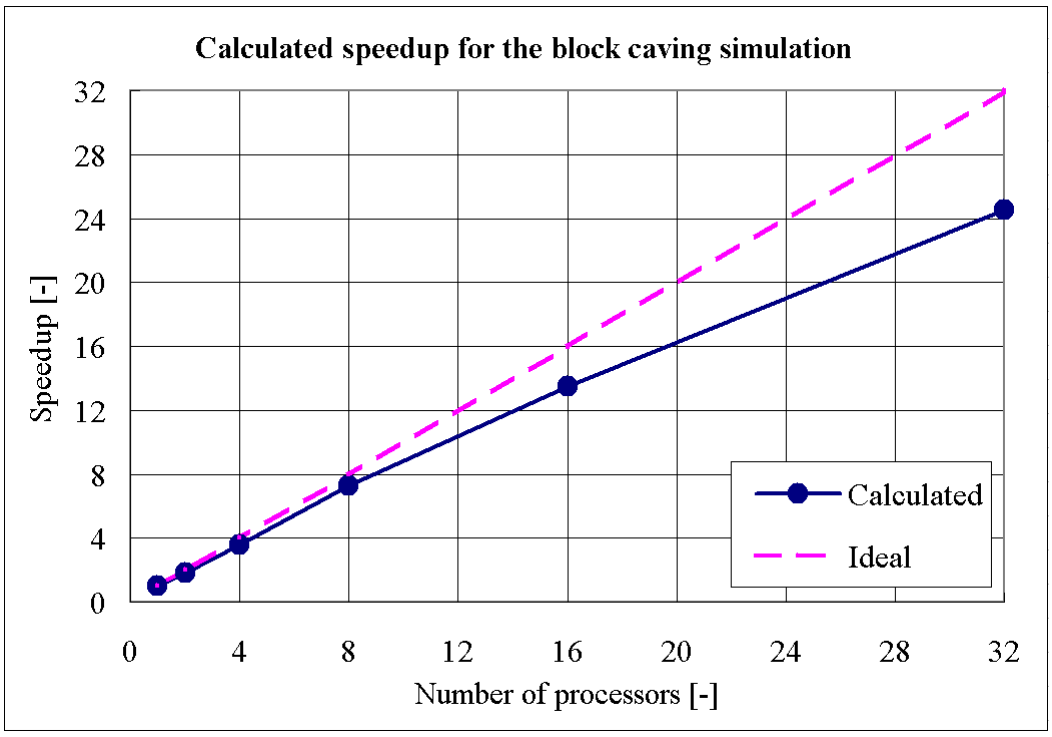


Figure 5.15: Calculated speedup for the block caving simulation.

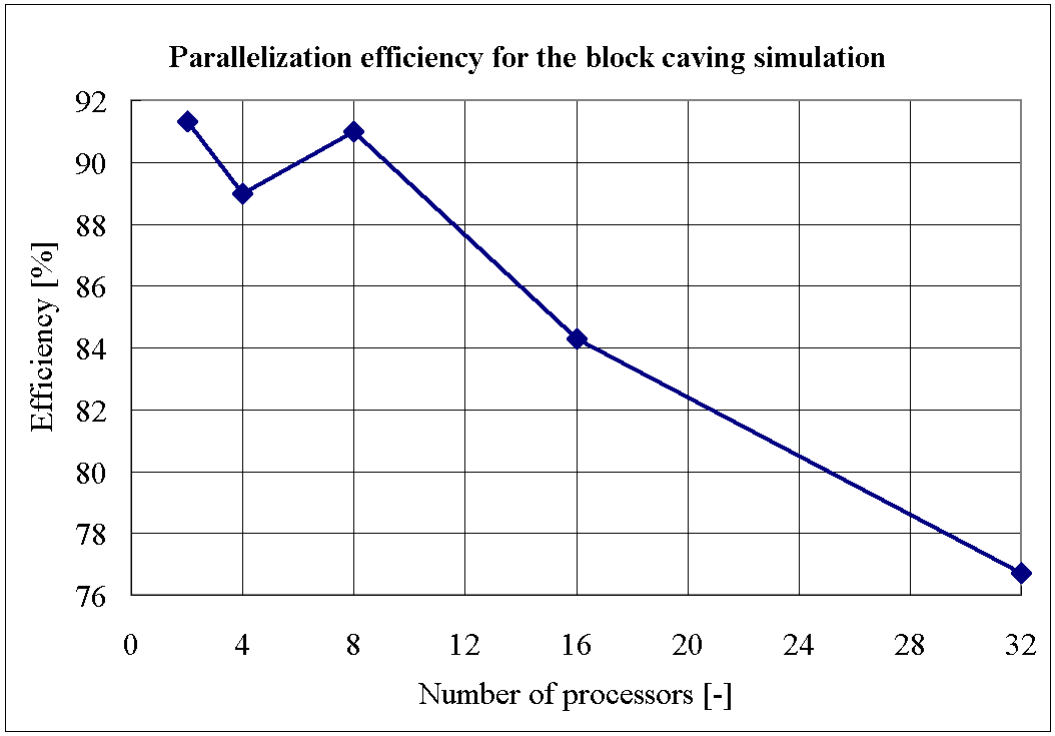


Figure 5.16: Calculated parallelisation efficiency for the block caving simulation.

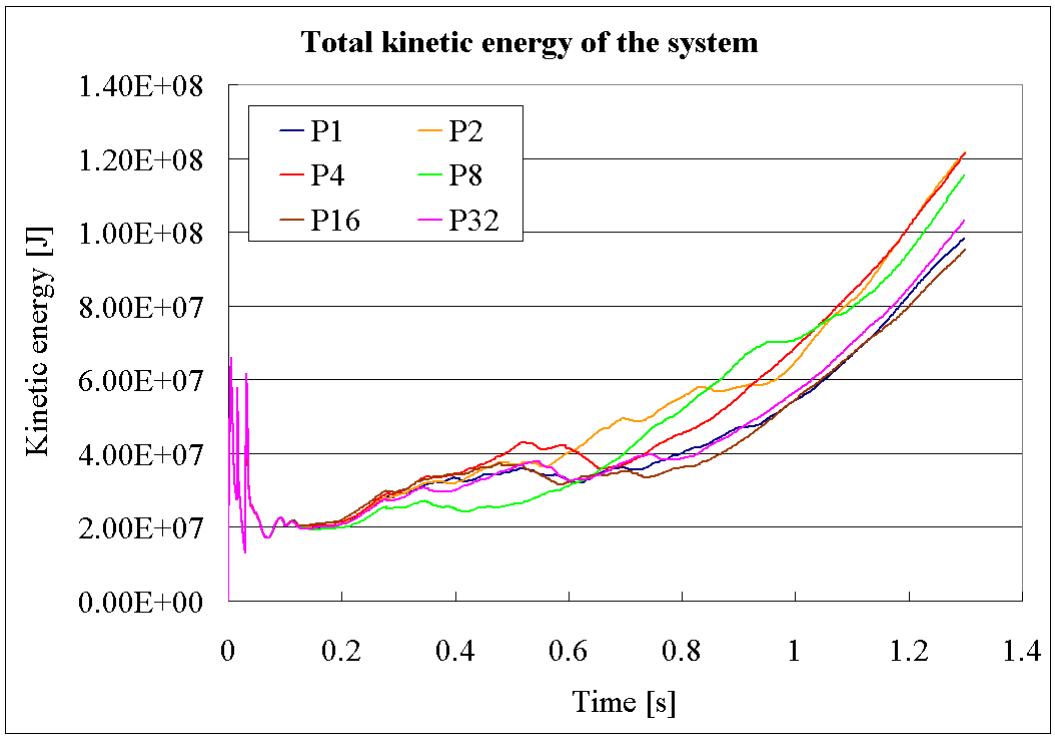


Figure 5.17: Total kinetic energy of the system for the whole simulation time.

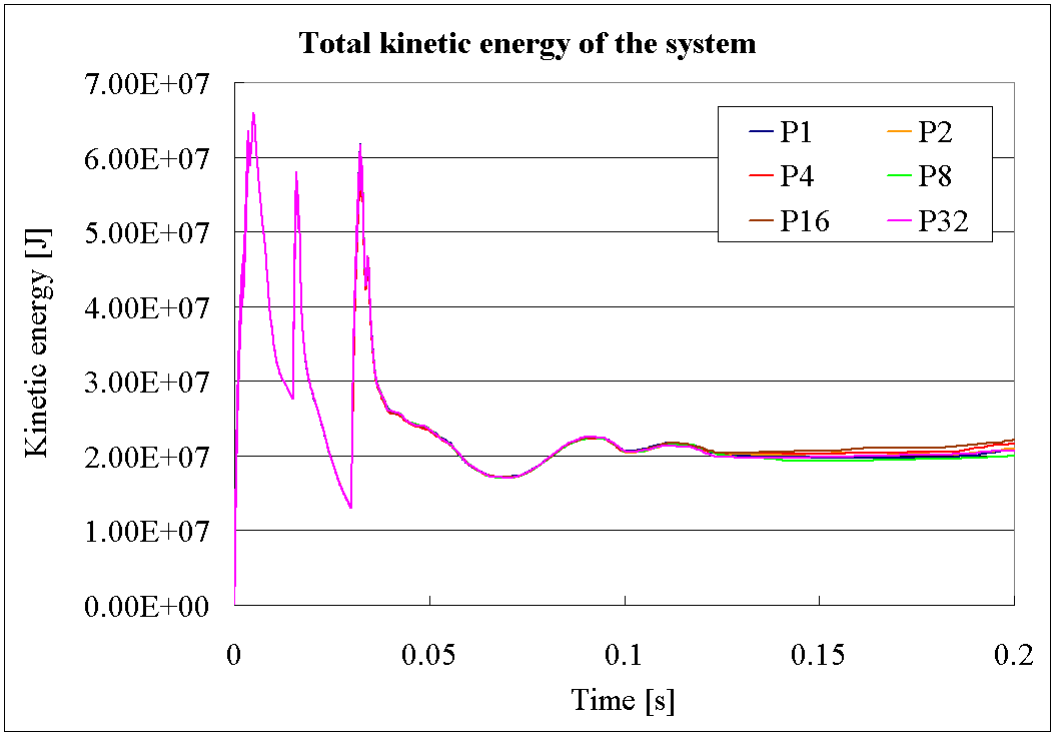


Figure 5.18: Total kinetic energy of the system from 0 s to 0.2 s.

Number of processors	CPU Time [s]	Speedup [-]	Efficiency [%]
1	63426	1	100
2	34727	1.83	91.32
4	17819	3.56	88.99
8	8713	7.28	90.99
16	4702	13.49	84.31
32	2584	24.55	76.71

Table 5.4: Recorded CPU times and calculated speedup and efficiency for the block caving simulation.

The second contributing reason for decreased performance with an increased number of processors is the migration of elements from one processor to another which creates a small imbalance. The imbalance is not big enough to trigger load balancing but it is enough to affect the performance of the whole system since the overall performance is given by the slowest processor.

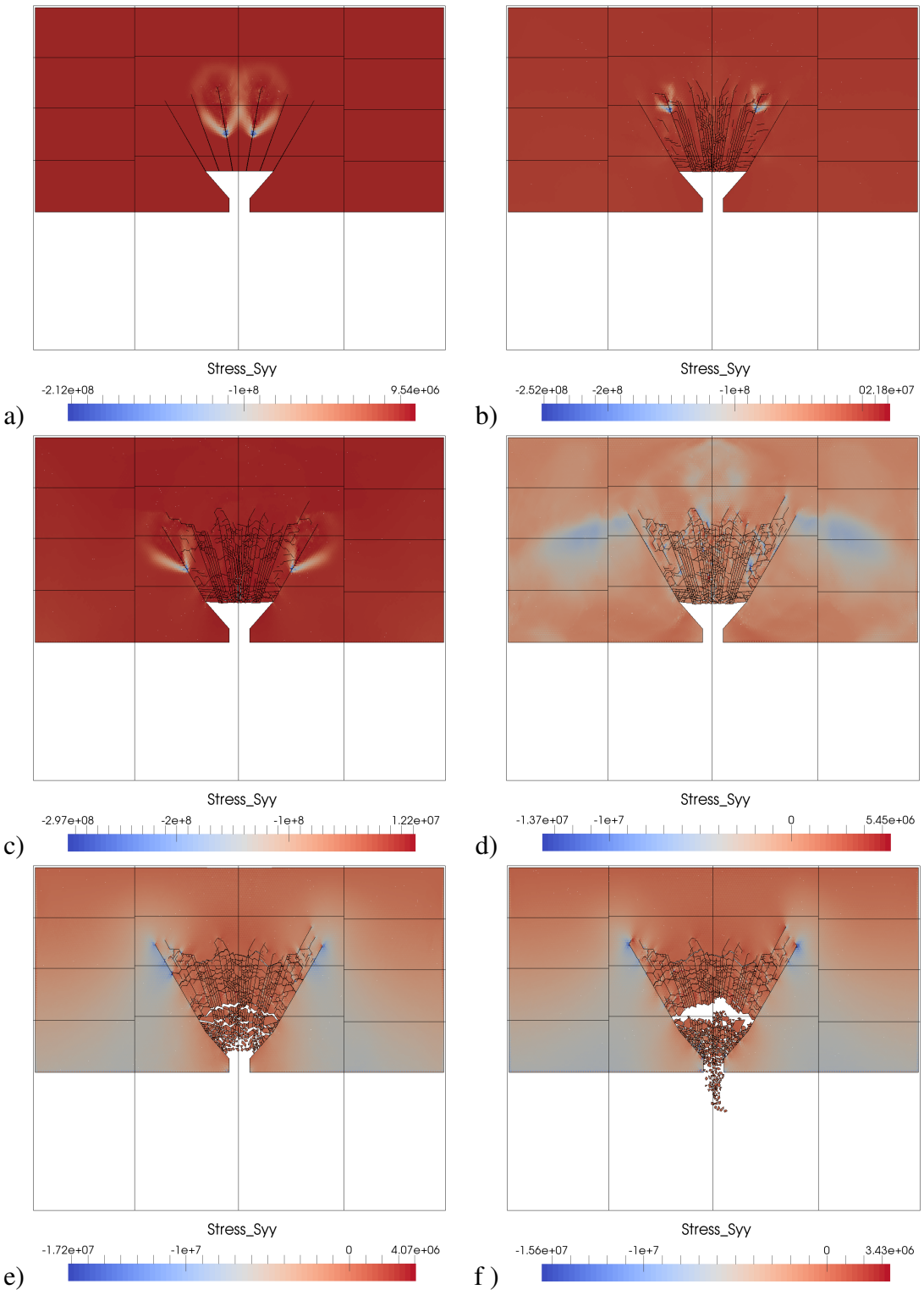


Figure 5.19: Block caving simulation sequence on 16 processors. a) Time 2.5 ms. b) Time 16 ms . c) Time 32.5 ms. d) Time 50 ms. e) Time 0.3 s. f) Time 0.8 s. Stress σ_y is in Pa.

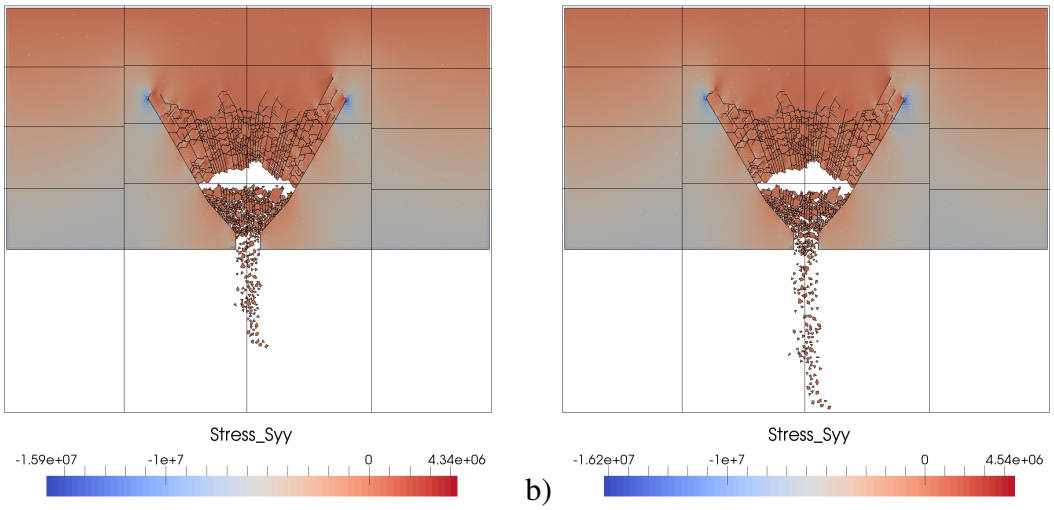


Figure 5.20: Block caving simulation sequence on 16 processors. a) Time 1.05 s. b) Time 1.3 s. Stress σ_y is in Pa.

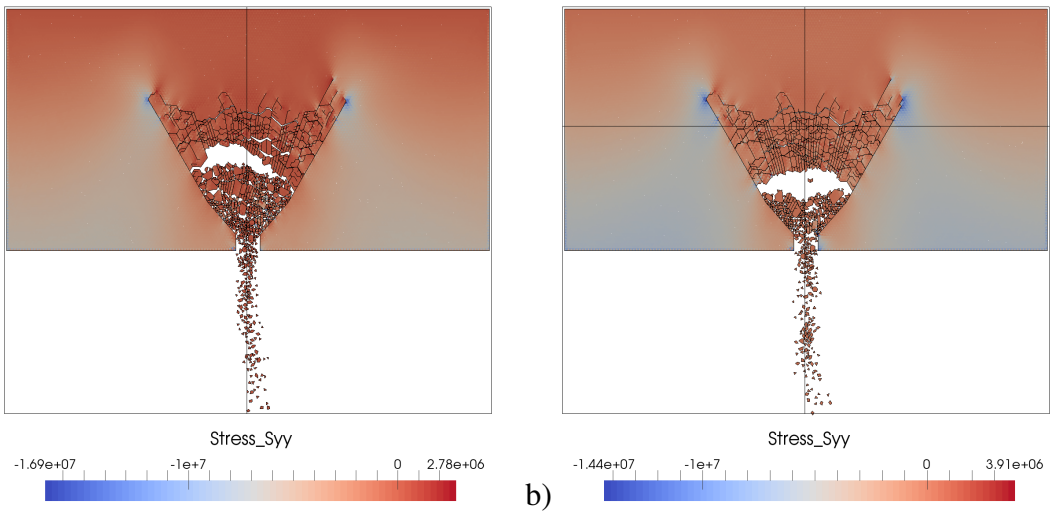


Figure 5.21: Results obtained for a block caving simulation at time 1.3 s for a) 2, b) 4 processors. Stress σ_y is in Pa.

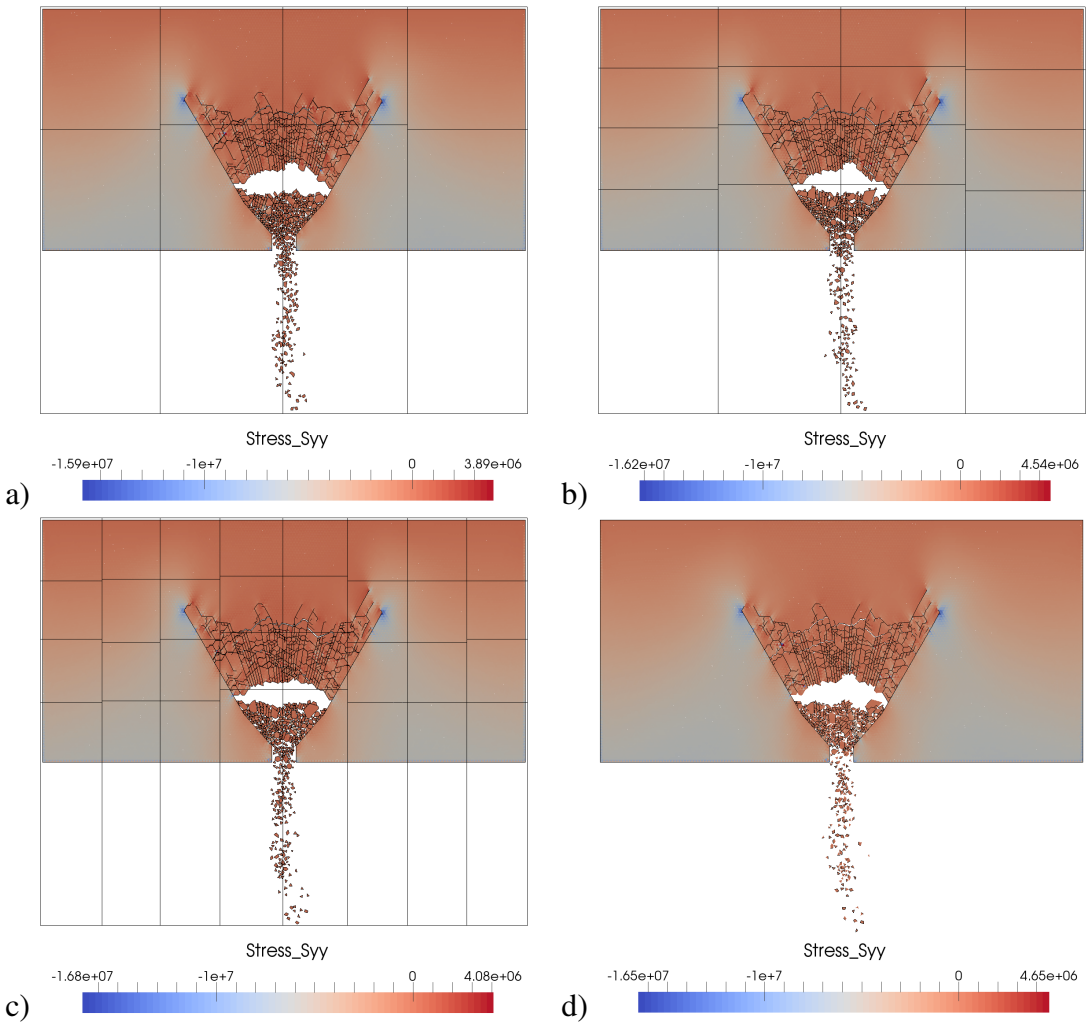


Figure 5.22: Results obtained for a block caving simulation at time 1.3 s for a) 8, b) 16, c) 32 processors and d) sequential run. Stress σ_y is in Pa.

The anomaly at 4 processors in Figure 5.16 can be caused by a higher initial imbalance. As explained in Chapter 5.1.3, the partitioning creates an initial imbalance which depends on the finite element mesh, distribution of discrete elements in the computational domain and the number of processors.

The imbalance for up to 8 processors is much smaller, since sizes of sub-domains containing moving particles are big enough and thus the migration of elements between sub-domains is very limited, see Figures 5.21 and 5.22. The performance could be improved by setting smaller maximum imbalance and also by setting a finer resolution of the load balancing grid.

The simulation sequence for the block caving simulation executed on 16 processors

is shown in Figures 5.19 and 5.20. Domain decompositions for 2, 4, 8, 16 and 32 processors as well as results for a sequential run are shown in Figures 5.21 and 5.22.

The fracture patterns as well as the general trend in the motion of particles obtained from parallel implementation of the FDEM code show a good correspondence with results obtained from a sequential FDEM code, see Figures 5.21 and 5.22. The comparison of total kinetic energies of the system for different numbers of processors shows a very good agreement up to approximately 0.2 s, see Figure 5.17. The first noticeable difference in the kinetic energy can be observed around 0.1 s (Figure 5.18) which is the time approximately 0.05 s after first particles start to fall. Due to the presence of rounding errors, the motion of particles is different on each processor, resulting in a different grow of kinetic energy.

The results obtained for the block caving simulation with given material properties suggest that the pressure amplitude of 2 GPa is enough to collapse the undercut area. Further studies should be made using the Y-Geo version of the code which is better suited for the rock mechanics problems and would give more realistic results.

5.3 Open Pit Slope

5.3.1 Introduction

Open pit mining is a surface mining technique where the rock is extracted from the ground of the open pit. It is usually used when the mineral or rock is near the surface or, when the presence of sand, gravel and similar materials make tunnelling unsafe. Design of an open pit slope is of great importance due to safety, ore recovery and financial return. Thus the stability of walls of the open pit mine is the major factor in their design. The stability is especially affected by the angle of the slope.

The open pit slope consists of intervals of several benches and a ramp with an access road[160] which is used by trucks to transport the mined rock.

5.3.2 Definition of the Problem

The stability of an open pit slope is numerically simulated on up to 4 processors. The material properties of the homogeneous rock (triangular elements) are as follows: Modulus of elasticity E = 31.68 GPa, Poisson's ratio $\nu = 0.2$, friction coefficient $f_C = 0.577$, density $\rho = 2650\,\mathrm{kg/m^3}$ and contact penalty is 11.7 GPa. The properties of joint elements are: shear strength 15 MPa, tensile strength 0.33 MPa, fracture en-

ergy 0.2 N/m and fracture penalty 11.7 GPa.[109] The time step is 5e-6 s and the total simulation time is 5 s. After the first fracturing occurs, the acceleration of gravity is linearly increased five times to speed up the fall of the fractured rock.

The aim of this simulation is to find the size of the maximum force which the ramp with an access road can withstand by increasing the force from 0 N to 7.848e6 N which corresponds with a load of 800 tonnes in the time interval from 0 s to 0.4 s (increase by 98.1 N in each time step). The force is applied on a second bench from the top, see Figure 5.23. The dimensions of the problem are 350×190 m and the dimensions of each bench are given in Figure 5.23 together with the initial mesh. The problem comprises 60368 triangular elements and 91702 joint elements.

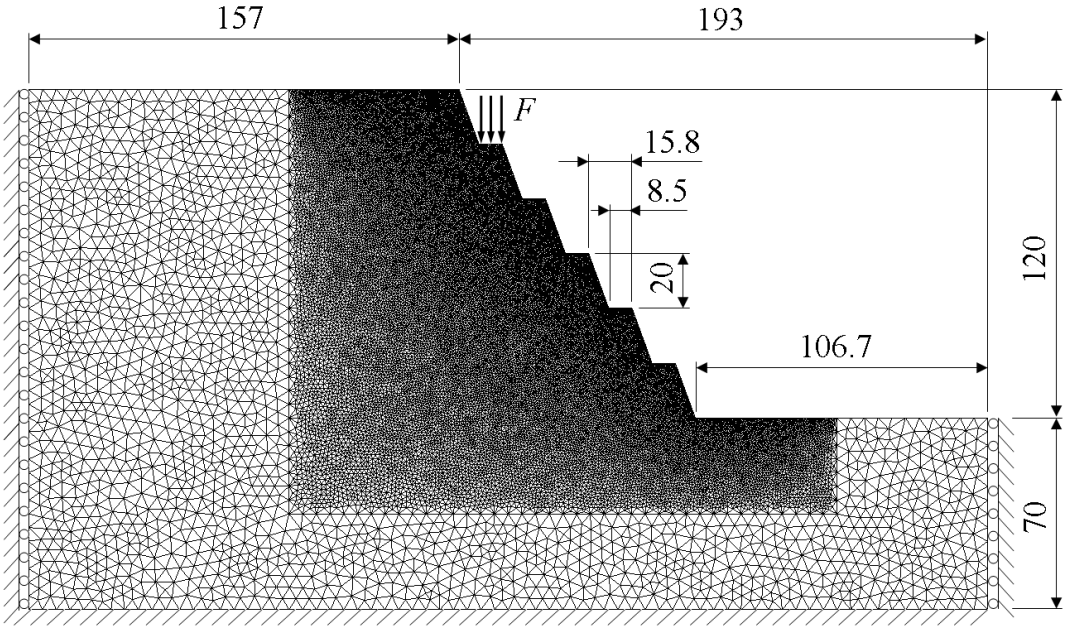


Figure 5.23: Initial mesh, boundary conditions and dimensions in metres for an open pit slope.

5.3.3 Results and Discussion

Recorded simulation times for different numbers of processors and calculated speedup and efficiency are summarised in Table 5.5. It should be noted that, due to the restrictions imposed by the cluster on the maximum job time, the simulation could not be run on the cluster. Thus, this problem is executed on a multicore PC with the following specifications: Dell Precission T3500 with Intel Xeon W3570 3.20 GHz (quad-core) and 12 GB of RAM.

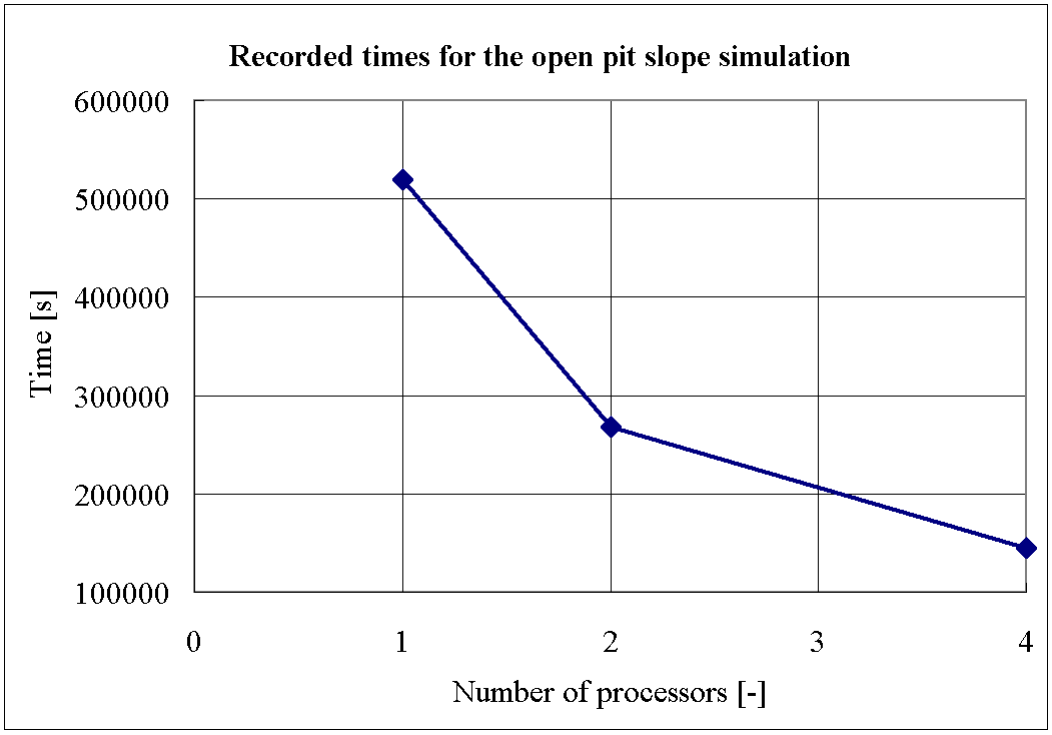


Figure 5.24: Recorded CPU times for the open pit slope simulation.

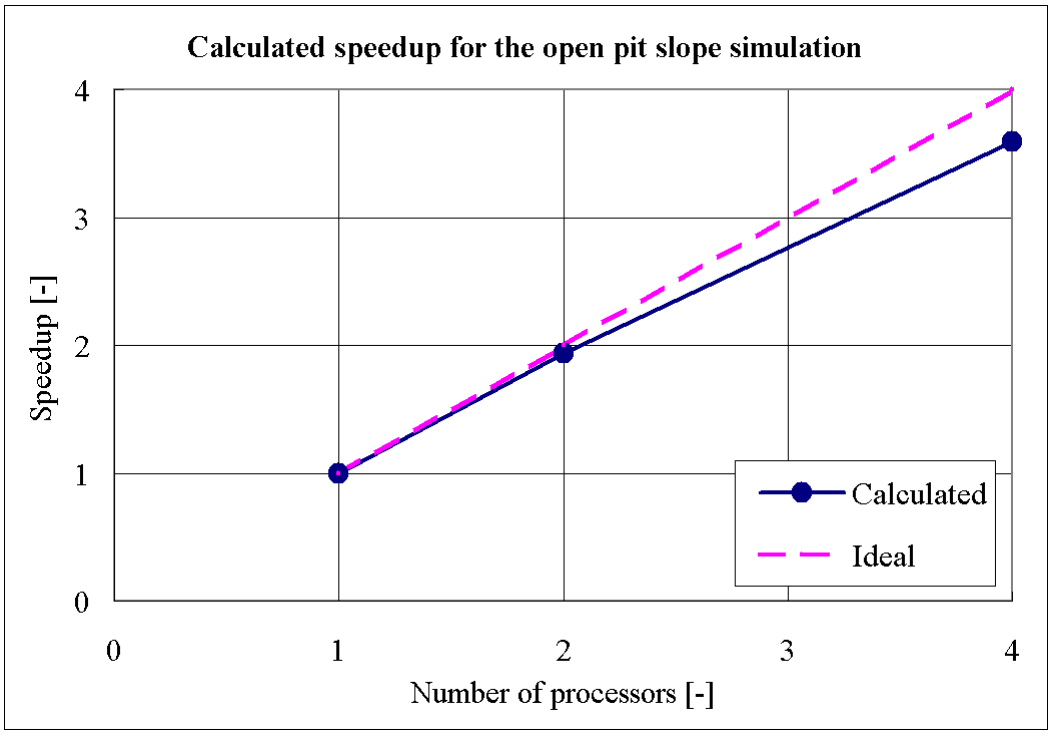


Figure 5.25: Calculated speedup for the open pit slope simulation.

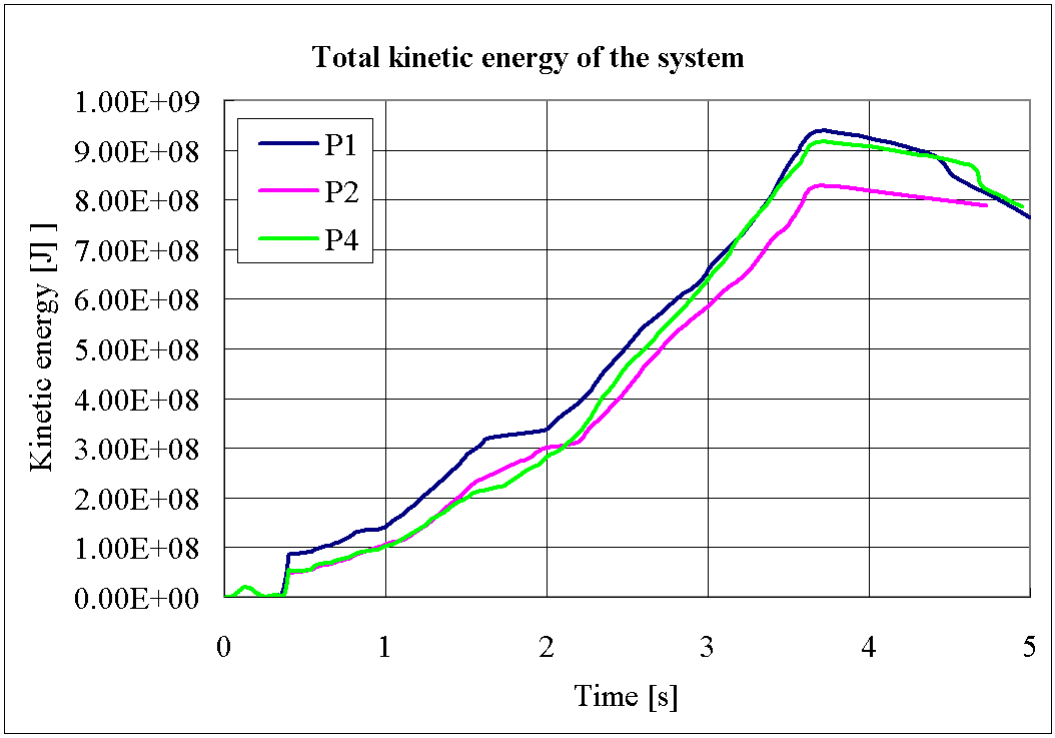


Figure 5.26: Total kinetic energy of the system for the whole simulation time.

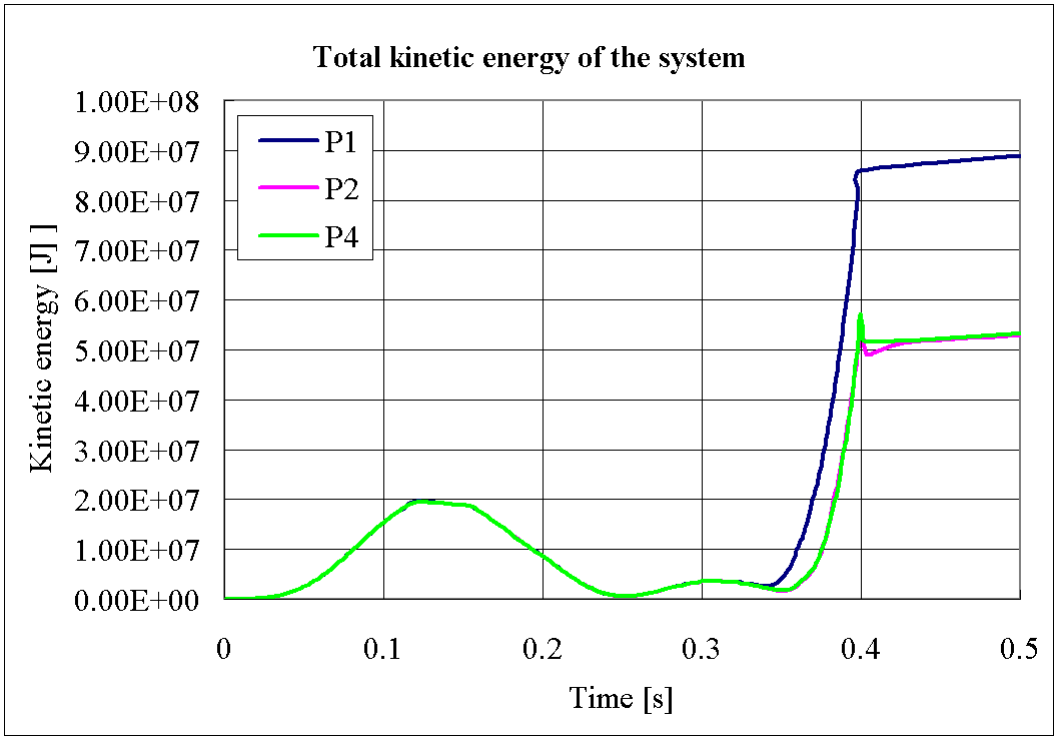


Figure 5.27: Total kinetic energy of the system from 0 s to 0.5 s.

Number of processors	CPU Time [s]	Speedup [-]	Efficiency [%]
1	519592	1	100
2	267854	1.94	96.99
4	144726	3.59	89.75

Table 5.5: Recorded CPU times, calculated speedup and efficiency for the open pit slope simulation.

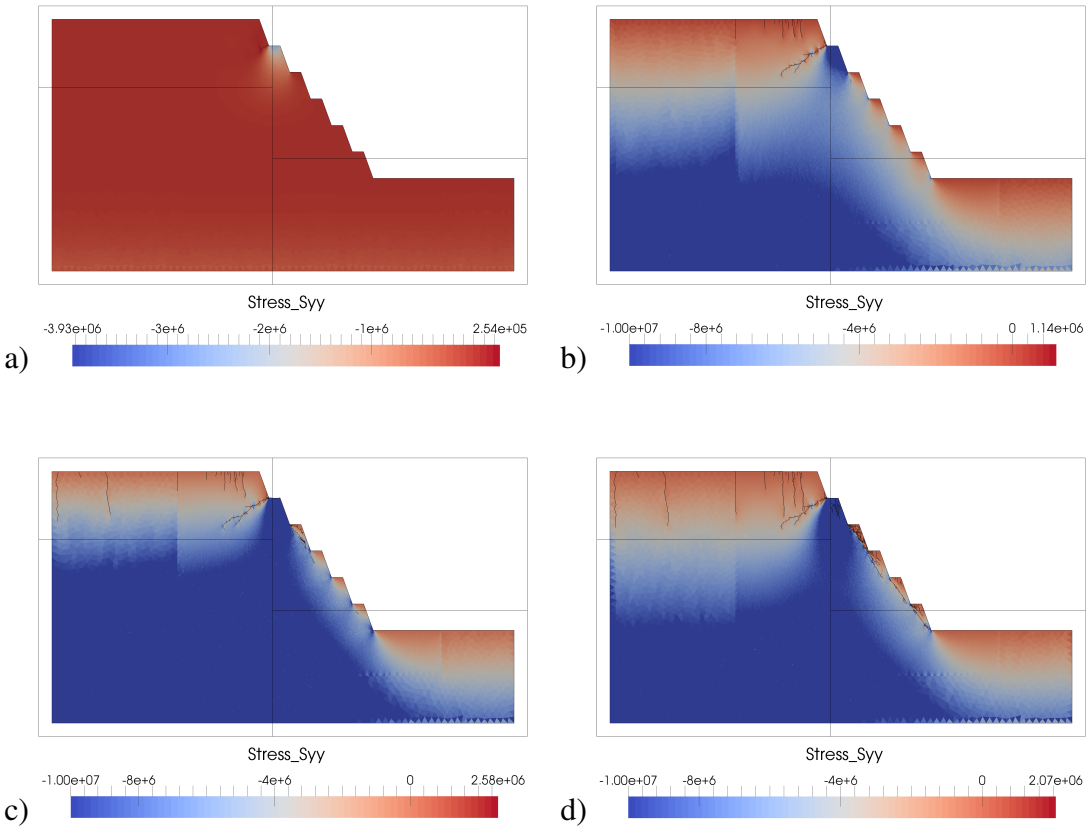


Figure 5.28: Open pit slope simulation sequence executed on 4 processors. a) Time 0.015 s. b) Time 0.15 s . c) Time 0.25 s. d) Time 0.35 s. Stress σ_y is in Pa.

Simulation times and calculated speedup for up to 4 processors are plotted in Figures 5.24 and 5.25. The main reason for the lower performance is the migration of significant number of elements from one processor to another creating a load imbalance. The communication overhead, in this case, does not play an important role since the problem is executed on a multicore PC, thus, the messages are not sent through

the network but copied from one memory location to another which is much faster. Further testing must be performed by defining a smaller maximum imbalance in order to trigger load balancing and improve the performance.

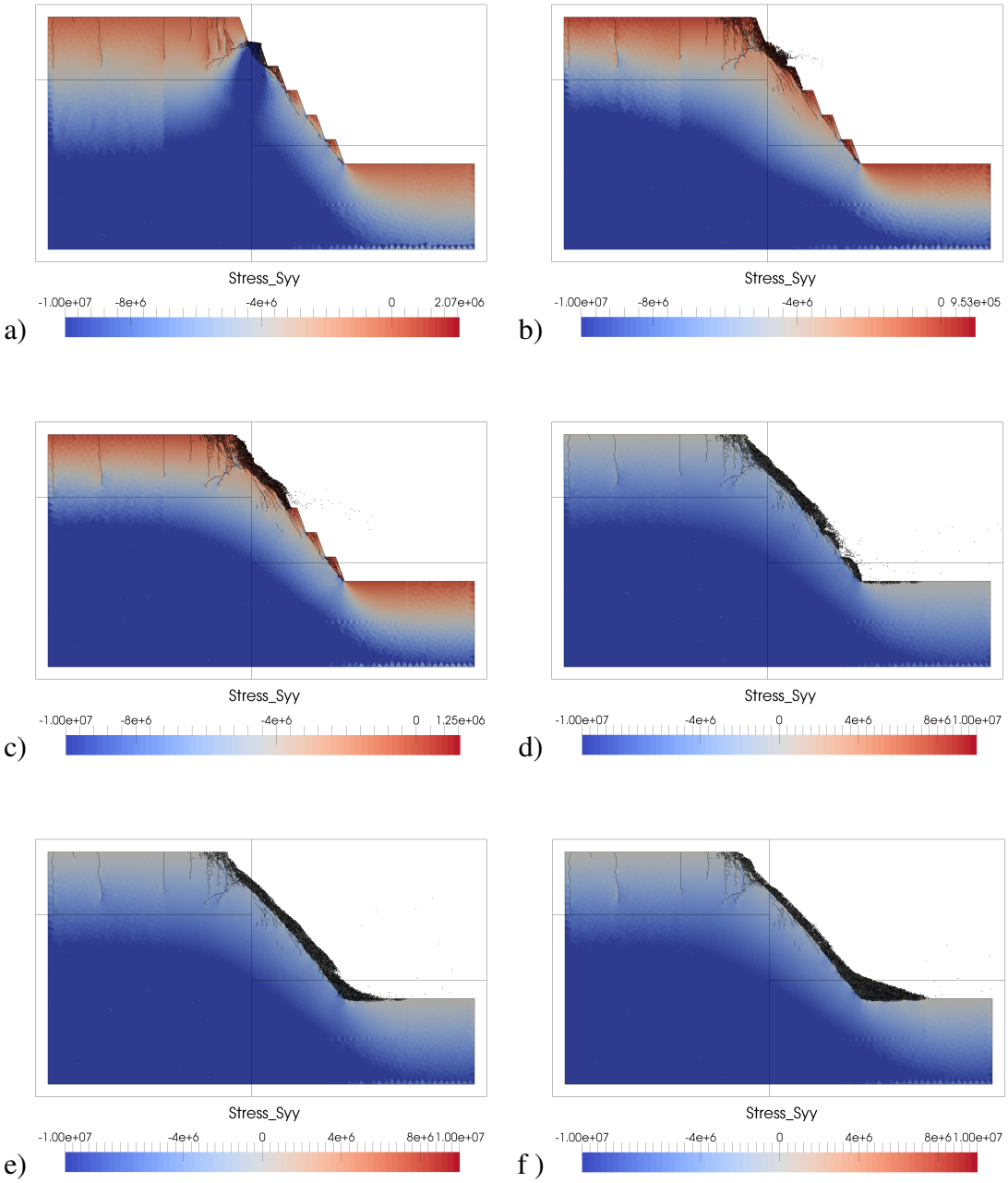
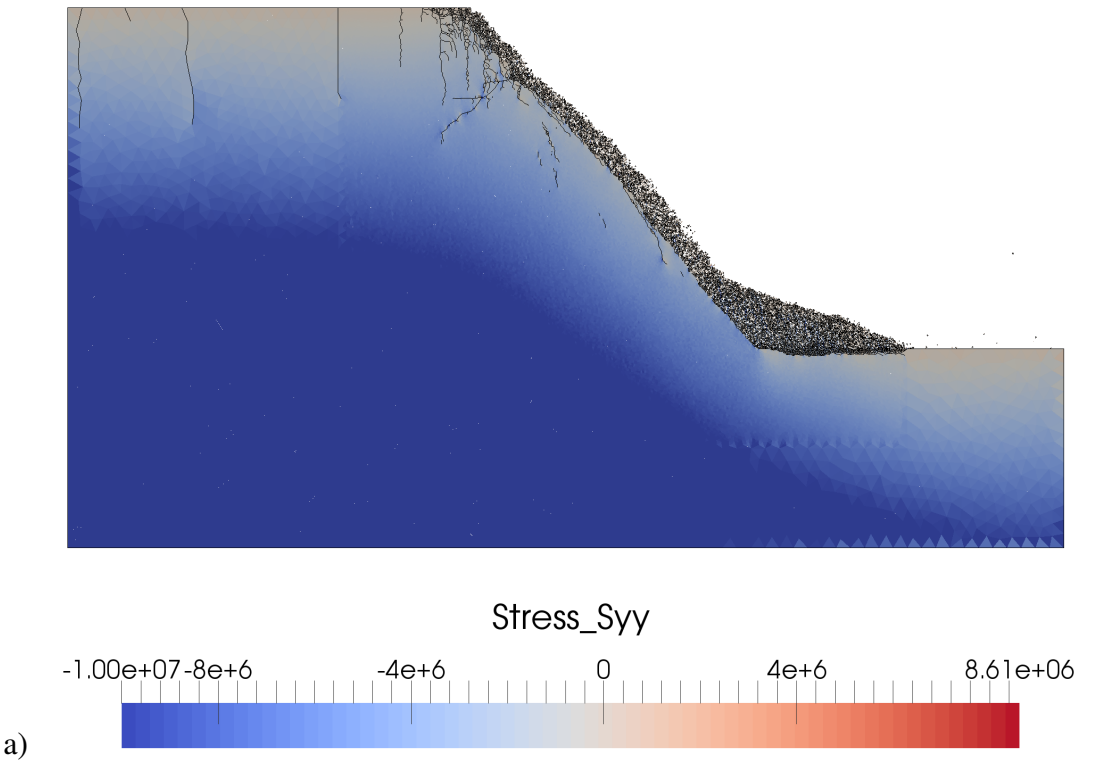
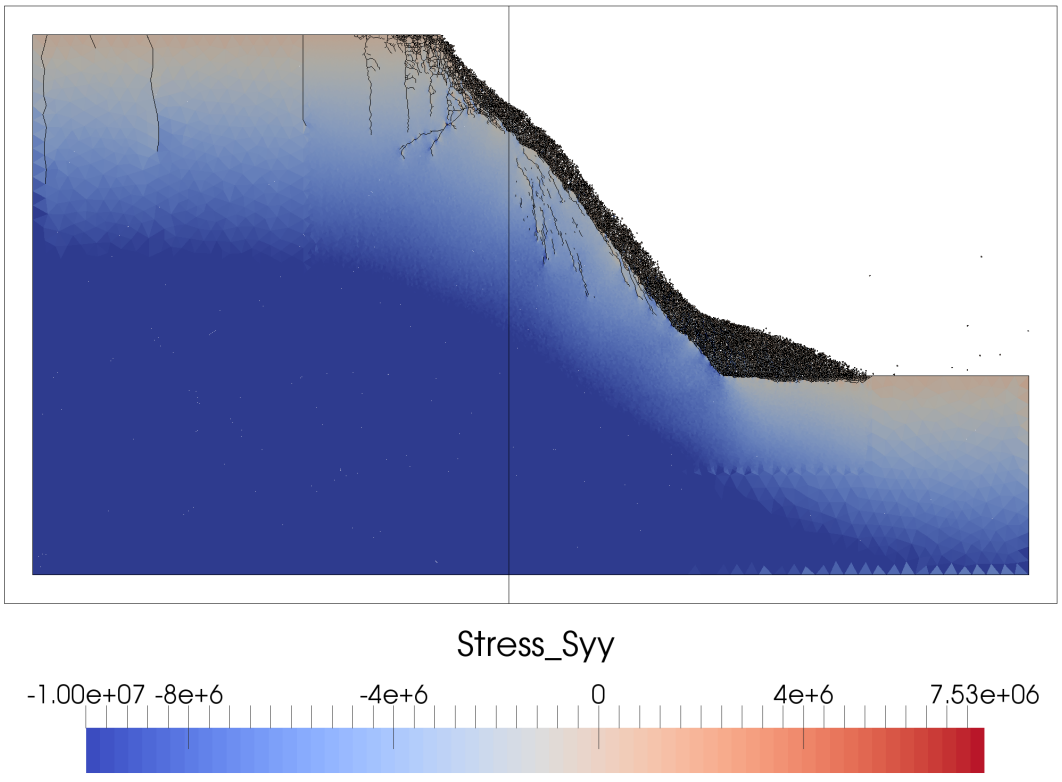


Figure 5.29: Open pit slope simulation sequence executed on 4 processors. a) Time 0.4 s. b) Time 0.75 s. c) Time 1.25 s. d) Time 2.5 s. e) Time 3.75 s. f) Time 5 s. Stress σ_y is in Pa.



a)



b)

Figure 5.30: Results obtained for an open pit slope simulation at time 5 s for a) sequential run, b) 2 processors. Stress σ_y is in Pa.

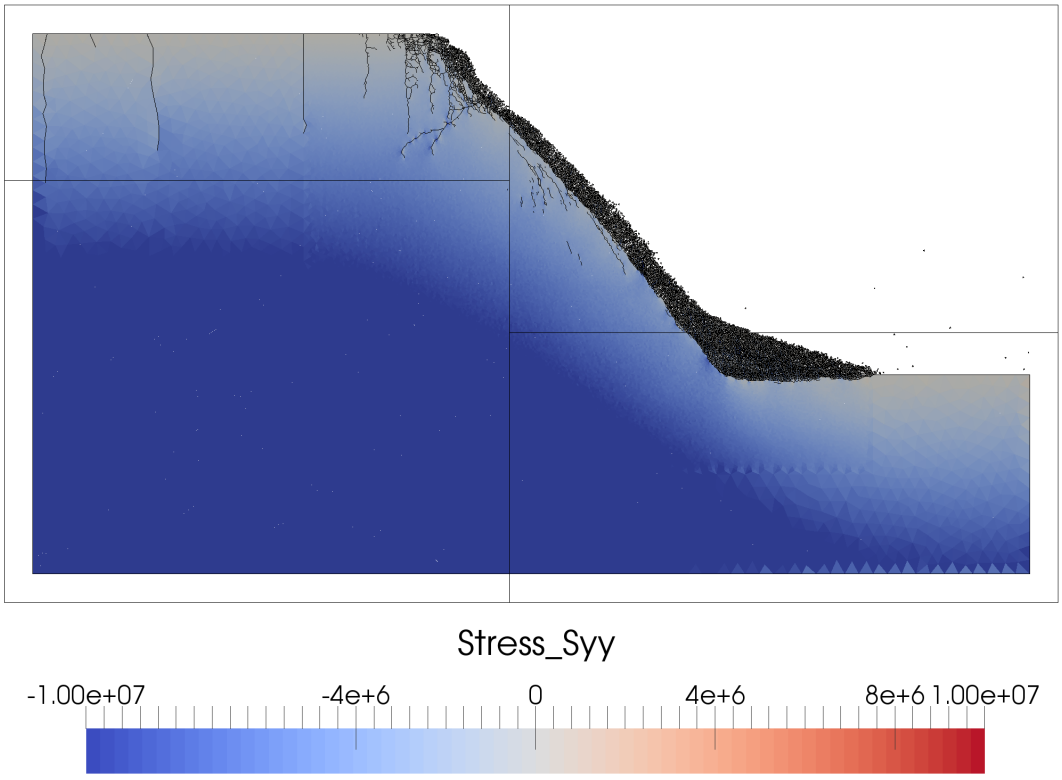


Figure 5.31: Results obtained for an open pit slope simulation at time 5 s for 4 processors. Stress σ_y is in Pa.

The motion sequence for the open pit slope simulation executed on 4 processors is shown in Figures 5.28 and 5.29. The results obtained from a sequential code as well as results and domain decomposition for 2 and 4 processors are shown in Figures 5.30 and 5.31.

The first crack appears at the base of the first bench from the top approximately at time 0.015 s which corresponds with a force of 294.3 kN (30 tonnes). Since this location has a sharp corner and introduces a singularity this can be neglected. The more significant fracture pattern appears around the time 0.15 s with a corresponding force of 2943 kN (300 tonnes). This should be the maximum load allowed on the bench decreased by a chosen safety factor.

The comparison of fracture patterns as well as the general trends in motion of particles (Figures 5.30 and 5.31) obtained for all three cases (sequential, 2 and 4 proces-

sors) reveal a good correspondence, thus further validating the parallel implementation of the FDEM code.

The total kinetic energy of the system for the whole simulation time is plotted in Figure 5.26. The general trend in evolution of the total kinetic energy in all cases shows a good correspondence. The first noticeable difference can be observed around the time 0.35 s, which corresponds with the time just before the first particles start to fall, see Figures 5.28d and 5.29a. Both fracture patterns and motion of particles are influenced by the presence of rounding errors resulting in differences in the total kinetic energies in all three cases.

5.4 Conclusions

The performance of the parallel implementation of the FDEM code compared with the performance of the sequential version of the code is good for all three numerical examples presented in this chapter. The speedup obtained for a Brazilian disc test, which is a quasi-static problem, shows a super-linear trend. The performance of the remaining two cases is lower due to the load imbalance. Further testing should be conducted by setting the maximum allowed imbalance to a smaller value than 20 % and by setting a finer resolution of the load balancing grid which would help to decrease the initial imbalance created during domain decomposition. To obtain more realistic results for the above numerical examples, the version of the FDEM code named Y-Geo should be employed.

The comparison of the general trend in the motion of particles, fracture patterns and total kinetic energies of the system in all three numerical examples, further validates proposed parallelisation solutions for FDEM and its implementation.

Chapter 6

CONCLUSIONS AND FUTURE WORK

Novel parallelisation solutions for the Combined Finite-Discrete Element Method (FDEM) have been developed. These have been implemented into the parallel version of the open source in-house FDEM code called Y2D. The summary of this thesis is presented and future research directions are suggested in this chapter.

Summary. Chapter 2 provided a short introduction to FDEM and methods of discontinua in general, as well as an introduction to parallel processing. The introduction of parallel processing started with an overview of available parallel architectures, past and present and provided a short description of Single Instruction-Multiple Data (SIMD) systems (vector processors and Graphics Processing Units) as well as Multiple Instruction-Multiple Data (MIMD) systems which include shared-memory systems and distributed-memory systems with the description of the commonly used interconnects. Main parallelisation languages/libraries currently used were introduced and a short description of basic MPI principles was provided. An overview of algorithms commonly used to perform domain decomposition and load balancing was presented and pros and cons for each algorithm were discussed. Finally some basic concepts of parallel computing, including the parallel performance measurements (speed-up, efficiency), were presented and the significance of floating point arithmetic in parallel processing was discussed.

A novel approach for parallelisation of FDEM in 2D aimed at clusters and desktop computers was described in detail in Chapter 3. Dynamic domain decomposition based parallelisation solvers covering all aspects of FDEM have been presented. The modified Recursive Coordinate Bisection (RCB) was employed to perform domain decomposition and load balancing. Both design and implementation of proposed parallel algorithms was explained in detail, including key parallelisation issues like communi-

cation, parallel input/output, migration of elements, re-partitioning and load balancing. A message-passing parallel programming model by using Message-Passing Interface (MPI) has been adopted.

Chapter 4 dealt with verification and performance tests of the proposed parallel algorithms. The parallel implementation of the FDEM code was tested on a following benchmark example: a box filled by 32400 discrete elements, each comprising 6 finite elements, moving across the box with initial velocity 100 m/s. The performance of the parallel code was good, considering the highly dynamical nature of the test case, and the speedup was around half the ideal speedup. The validity of the developed parallel solvers was confirmed by comparing a general trend in motion of discrete elements, as well as by a comparison of total kinetic energy of the system obtained from a sequential and a parallel version of FDEM code.

Some application examples of the parallel FDEM code chosen from the field of rock mechanics were presented in Chapter 5. Performance and scalability of the parallel FDEM code is further studied on the following numerical examples:

- Brazilian disc test is a common laboratory test designed for indirect measurement of the tensile strength of brittle materials. The results showed a good correspondence with previously published results, even though the original version of the Y2D code is used. Y-Geo, specifically designed for applications in the rock mechanics field, was recommended to conduct further research. The calculated speedup had a super-linear trend, thus the efficiency of the parallel code was over 100 % on up to 32 processors.

- Block caving simulation was employed to find the value of the pressure amplitude which would be big enough to collapse the undercut area. The recommended pressure amplitude was 2 GPa. The speedup had a linear trend up to 8 processors and, for the higher numbers of processors, the parallel efficiency decreased due to the small imbalance, as well as the decreasing ratio between computation and parallelisation overhead.

- Open pit slope simulation was used to find the maximum allowed load on a bench. The force found was 2943 kN corresponding with 300 tonnes, thus allowing the safe use as an access road to the mine. The calculated speedup was good and the parallel efficiency was around 90 % for 4 processors and 97 % for 2 processors. Lower efficiency was found to be due to load imbalance. Means to rectify the problem were outlined.

Conclusions. The implementation of the proposed novel parallelisation solutions for FDEM has been tested on chosen numerical examples. Comparison of results (general trend in motion of discrete elements, fracture pattern and total kinetic energy of the system), obtained for examples presented in Chapter 4 and Chapter 5 by using both sequential and parallel versions of FDEM code, verified the validity of the developed parallel FDEM solvers. The performance of the parallel version of FDEM code for different types of numerical examples (from a quasi-static example to a highly dynamic one) ranged from a super-linear speedup to approximately half the ideal speedup. Nevertheless, based on results obtained, the parallel implementation of FDEM code should scale well to enable solving of large scale and grand challenge FDEM simulations.

Future work. Possibilities for further research are presented and discussed in the remainder of this chapter.

The presented implementation of the developed parallel FDEM solvers was tested only on numerical examples of moderate sizes. In order to enable simulation of large scale and grand challenge FDEM problems, the issues with parallel input and output must be addressed. For instance, a simulation comprising 7 million elements requires an input and output files of around 1 GB. In order to solve problems with one hundred million elements or even more, the required input/output file would have to be enormous (hundreds of GB).

The solution for the input file issue lies in generating multiple input files for each processor separately. Thus the initial domain decomposition would be performed during the input file generation, resulting in a number of input files of manageable sizes. Practically all MPI implementations nowadays support input/output operations performed by all processors comprising the simulation.

The solution for the output file issue is not so straightforward. A decision must be made if one large file is more manageable than each processor writing its own output file. Unlike an input file which is read by the program only once, the output must be written many times during the simulation. Then the output for the whole simulation would have size in order of terabytes or even petabytes. Thus, some efficient compression could be utilised. The last problem in this area lies in actually post-processing these large output files. This would require enormous computational power. Graphics Processing Units (GPUs) are the best suited for these kind of problems. This would require GPU-based parallelisation of a post-processor if no suitable parallel post-processor is available.

The developed parallel solvers are designed for 2D version of the FDEM code. The 3D version of the FDEM code became available in recent years. Consequently,

the next logical step is to extend the developed parallel solvers into 3D. This task should be relatively straightforward, since the RCB algorithm can be directly applied to any number of dimensions. The 3D parallelisation would require extending the classification of elements based on their geometric location and also changing the 2D communication pattern into 3D.

The performance of the parallel implementation can be improved by employing non-blocking communication, thus sending messages while performing other computations, and each processor would have to check from time to time if the message is available for receiving. The use of non-blocking communication in FDEM simulation is not a trivial task, since the simulation should be synchronized across all processors from time to time, preferably with each time step. The only way to avoid an explicit call of MPI synchronization function in each time step would be to ensure an equal workload on all processors. This would require executing the load balancing routine quite often.

Finally, as mentioned in Chapter 5, several modified versions of the original FDEM code exist in both 2D and 3D. These include Y-Geo[110] developed for use in rock mechanics, Y-Nano[163] designed for molecular dynamics simulations and a coupled FDEM/CFD code which has been partially parallelised to enable a large scale simulation of red blood cells.[207] That being so, it is desirable to port the developed parallelisation solvers into these codes in order to solve large scale problems as well as into possible future codes for the coupling of FDEM with other methods. For instance, a relatively new method for fluid dynamics is a Lattice Boltzmann (LB) method,[179] which uses Boltzmann equations to solve Navier-Stokes equations on a predefined lattice. LB has been already coupled with DEM by Feng et al.[44] and Han et al.[60] in order to solve fluid-particle interactions. Similarly to FDEM the equations are local and thus the coupling of both methods and its parallelisation is a promising future research direction.

References

[1] ; PACS Training Group (Veranst.): *Introduction to MPI*. On-Line. `http://www.rochester.edu/college/psc/thestarlab/help/MPI_Course.pdf`. Version: 2001. – Accessed on 07 Aug 2014

[2] *NVIDIA CUDA Compute Unified Device Architecture: programming guide 2.0*. NVIDIA Corporation, 2008

[3] *cplusplus.com*. `http://www.cplusplus.com/`. Version: 2013. – Accessed 29th October 2013

[4] AKTULGA, H. M. ; FOGARTY, J. C. ; PANDIT, S. A. ; GRAMA, A. Y.: Parallel Reactive Molecular Dynamics: Numerical Methods and Algorithmic Techniques. In: *Parallel Computing* 38 (2012), S. 245–259

[5] AMDAHL, G. M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: *Proceedings of the American Federation of Information Processing Societies Conference* Bd. 30, 1967, S. 483–485

[6] AMRITKAR, A. ; DEB, S. ; TAFTI, D. : Efficient Parallel CFD-DEM Simulations Using OpenMP. In: *Journal of Computational Physics* 256 (2014), S. 501–519

[7] APOSTOLOU, K. ; HRYMAK, A. N.: Discrete Element Simulation of Liquid-Particle Flows. In: *Computers and Chemical Engineering* 32 (2008), S. 841–856

[8] BARONE, L. M. ; SIMONAZZI, R. ; TENENBAUM, A. : Molecular Dynamics on APE100. In: *Computer Physics Communications* 90 (1995), S. 44–58

[9] BELIKOV, E. ; DELIGIANNIS, P. ; TOTOO, P. ; ALJABRI, M. ; LOIDL, H. W.: A Survey of High-Level Parallel Programming Models / Heriot-Watt University, Edinburgh. 2013. – Forschungsbericht

[10] BERENBAUM, R. ; BRODIE, I. : Measurement of the tensile strength of brittle materials. In: *British Journal of Applied Physics* 10 (1959), Nr. 281–287

[11] BERGER, M. J. ; BOKHARI, S. H.: A Partitioning Strategy for Nonuniform Problems on Multiprocessors. In: *IEEE Transactions on Computers* C-36 (1987), S. 570–580

[12] BERGER, R. ; KOHLMEYER, A. ; KLOSS, C. : Towards Parallelization of LIGGGHTS Granular Force Kernels with OpenMP. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[13] BIRD, G. A.: *Molecular Gas Dynamics and the Direct Simulation of Gas Flows*. Oxford Sience Publications, 1994

[14] BLAKE, G. ; DRESLINSKI, R. G. ; MUDGE, T. : A Survey of Multicore Processors. In: *IEEE Signal Processing Magazine* 26 (2009), S. 26–37

[15] BOILLAT, J. E. ; BRUGE, F. ; KROPF, P. G.: A Dynamic Load-Balancing Algorithm for Molecular Dynamics Simulation on Multi-Processor Systems. In: *Journal of Computational Physics* 96 (1991), S. 1–14

[16] BORKAR, S. ; CHIEN, A. : The Future of Microprocessors. In: *Communications of the ACM* 5 (2011), S. 67–77

[17] BUI, T. ; JONES, C. : A Heuristic for Reducing Fill in Sparse Matrix Factorization. In: *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM*, 1993, S. 445–452

[18] CHANDRA, R. ; DAGUM, L. ; KOHR, D. ; MAYDAN, D. ; MCDONALD, J. ; MENON, R. : *Parallel programming in OpenMP*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – ISBN 1–55860–671–8

[19] CHAPLOT, S. L.: Parallelization in Classical Molecular Dynamics Simulation and Applications. In: *Computational Materials Science* 37 (2006), S. 146–151

[20] CHAPMAN, B. ; JOST, G. ; PAS, R. van d.: *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008

[21] CHEN, X. ; CHAN, A. H. C. ; YANG, J. : A case study of impact on glass using the combined Finite-Discrete Element method. In: MUNJIZA, A. (Hrsg.):

Discrete Element Methods, Simulatios of Discontinua: Theory and Application, 2010, S. 465–469

[22] CHRISOCHOIDES, N. : *Multithreaded Model for Dynamic Load Balancing Parallel Adaptive PDE Computations*. ICASE Report 95-83, ICASE, NASA Langley Research Center, Hampton, VA 23681-0001, December 1995

[23] CLARK, T. W. ; HANXLEDEN, R. von ; MCCAMMON, J. A. ; SCOTT, L. R.: Parallelizing Molecular Dynamics Using Spatial Decomposition. In: *Proceedings of the Scalable High-Performance Computing Conference*, 1994

[24] CLARK, T. W. ; MCCAMMON, J. A.: Parallelization of a Molecular Dynamics Non-Bonded Force Algorithm for MIMD Architecture. In: *Computers Chem.* 14 (1990), S. 219–224

[25] CLEARY, P. W. ; SAWLEY, M. L.: DEM Modelling of Industrial Granular Flows: 3D Case Studies and the Effect of Particle Shape on Hopper Discharge. In: *Applied Mathematical Modelling* 26 (2002), S. 89–111

[26] CORNWELL, C. F. ; WILLE, L. T.: Parallel Molecular Dynamics Simulations for Short-Ranged Many-Body Potentials. In: *Computer Physics Communications* 128 (2000), S. 477–491

[27] COUGNY, H. de ; DEVINE, K. ; FLAHERTY, J. ; LOY, R. ; OZTURAN, C. ; SHEPHARD, M. : Load Balancing for the Parallel Adaptive Solution of Partial Differential Equations. In: *Appl. Numer. Math.* 16 (1994), S. 157–182

[28] CUNDALL, P. A.: A computer model for simulating progressive large scale movements in block rock systems. In: *Proceedings of the symposium of international society of rock mechanics*, 1971

[29] CUNDALL, P. A. ; STRACK, O. D. L.: A discrete numerical model for granular assemblies. In: *Geotechnique* 29 (1979), Nr. 1, S. 47–65

[30] CYBENKO, G. : Dynamic Load Balancing for Distributed Memory Multiprocessors. In: *Journal of Parallel and Distributed Computing* 7 (1989), S. 279–301

[31] DENG, Y.-F. ; MCCOY, R. A. ; MARR, R. B. ; PEIERLS, R. F. ; YASAR, O. : Molecular Dynamics on Distributed-Memory MIMD Computers with Load Balancing. In: *Applied Mathematical Letters* 8 (1995), S. 37–41

[32] DEVINE, K. ; FLAHERTY, J. : Parallel Adaptive HP-refinement Techniques for Conservation Laws. In: *Appl. Numer. Math.* 20 (1996), S. 367–386

[33] DIEKMANN, R. ; FROMMER, A. ; MONIEN, B. : Efficient Schemes for Nearest Neighbor Load Balancing. In: *Parallel Computing* 25 (1999), S. 789–812

[34] DJINEVSKI, L. ; MISHKOVSKI, I. ; TRAJANOV, D. : Accelerating Clustering Coefficient Calculations on a GPU Using OPENCL. In: *ICT Innovations 2010* (2011), S. 276–285

[35] DOMÍNGUEZ, J. M. ; CRESPO, A. J. C. ; VALDEZ-BALDERAS, D. ; ROGERS, B. D. ; GÓMEZ-GESTEIRA, M. : New Multi-GPU Implementation for Smoothed Particle Hydrodynamics on Heterogeneous Clusters. In: *Computer Physics Communications* 184 (2013), S. 1848–1860

[36] DOWDING, C. H. ; DMYTRYSHYN, O. ; BELYTSCHKO, T. B.: Parallel Processing for a Discrete Element Program. In: *Computers and Geotechnics* 25 (1999), S. 281–285

[37] DUFF, I. S.: Architectures and Systems. In: *Computer Physics Reports* 11 (1989), S. 1–20

[38] EISENHAUER, G. ; SCHWAN, K. : Design and Analysis of a Parallel Molecular Dynamics Application. In: *Journal of Parallel and Distributed Computing* 35 (1996), S. 76–90

[39] ENBODY, R. ; PURDY, R. ; SEVERANCE, C. : Dynamic Load Balancing. In: *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, SIAM*, 1995, S. 645–646

[40] ESSELINK, K. ; HILBERS, P. A. J.: Efficient Parallel Implementation of Molecular Dynamics on a Toroidal Network. Part II. Multi-Particle Potentials. In: *Journal of Computational Physics* 106 (1993), S. 108–114

[41] ESSELINK, K. ; SMIT, B. ; HILBERS, P. A. J.: Efficient Parallel Implementation of Molecular Dynamics on a Toroidal Network. Part I. Parallelizing Strategy. In: *Journal of Computational Physics* 106 (1993), S. 101–107

[42] FATTEBERT, J.-L. ; RICHARDS, D. F. ; GLOSLI, J. N.: Dynamic Load Balancing Algorithm for Molecular Dynamics Based on Voronoi Cell Domain De-

compositions. In: *Computer Physics Communications* 183 (2012), S. 2608–2615

[43] FELDMAN, J. ; BONET, J. : Dynamic refinement and boundary contact forces in SPH with applications in fluid flow problems. In: *International Journal for Numerical Methods in Engineering* 72 (2007), Nr. 3, S. 295–324. – ISSN 1097–0207

[44] FENG, Y. T. ; HAN, K. ; OWEN, D. R. J.: Coupled lattice Boltzmann method and discrete element modelling of particle transport in turbulent fluid flows: Computational issues. In: *International Journal for Numerical Methods in Engineering* 72 (2007), S. 1111–1134

[45] FLAHERTY, J. ; LOY, R. ; SHEPHARD, M. ; SZYMANSKI, B. ; TERESCO, J. ; ZIANTZ, L. : Adaptive Local Refinement with Octree Load-Balancing for the Parallel Solution of Three-Dimensional Conservation Laws. In: *J. Parallel Distrib. Comput.* 47 (1998), S. 139–152

[46] FLYNN, M. J.: Very High-Speed Computing Systems. In: *Proceedings of the IEEE* 54 (1966), S. 1901–1909

[47] FORUM, M. P. I.: *MPI: A Message-Passing Interface Standard Version 3.0*. `http://www.mpi-forum.org/docs/docs.html`. Version: September 2012. – Acceced 28 Feb 2013

[48] FRENNING, G. : An Efficient Finite/Discrete Element Procedure for Simulating Compression of 3D Particle Assemblies. In: *Comput. Methods Appl. Mech. Engrg.* 197 (2008), S. 4266–4272

[49] FRIEDRICHS, M. S. ; EASTMAN, P. ; VAIDYANATHAN, V. ; HOUSTON, M. ; LEGRAND, S. ; BEBERG, A. L. ; ENSIGN, D. L. ; BRUNS, C. M. ; PANDE, V. S.: Accelerating Molecular Dynamic Simulation on Graphics Processing Units. In: *Journal of Computational Chemistry* 30 (2009), S. 864–872

[50] GADIROV, V. ; EPPELBAUM, L. : Density-thermal dependence of sedimentary associations call to reinterpreting detailed gravity surveys. In: *Annals of Geophysics* 58 (2015), S. 1–6

[51] GEIST, A. ; BEGUELIN, A. ; DONGARRA, J. ; JIANG, W. ; MANCHEK, R. ; SUNDERAM, V. : *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994

[52] GOLDBERG, D. : What Every Computer Scientist Should Know About Floating-Point Arithmetic. In: *ACM Computing Surveys* 23 (1991), S. 5–48

[53] GOPALAKRISHNAN, P. ; TAFTI, D. : Development of Parallel DEM for the Open Source Code MFIX. In: *Powder Technology* 235 (2013), S. 33–41

[54] GOVENDER, N. ; GLEDHILL, I. ; KOK, S. ; WILKE, N. : GPU-Based Discrete Element Rigid Body Transport. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[55] GOVENDER, N. ; WILKE, D. N. ; KOK, S. ; ELS, R. : Development of a Convex Polyhedral Discrete Element Simulation Framework for NVIDIA Kepler Based GPUs. In: *Journal of Computational and Applied Mathematics* 270 (2014), S. 386–400

[56] In: GREWE, D. ; OBOYLE, M. : *A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL*. Bd. 6601. Springer Berlin Heidelberg, 286–305

[57] GRIEBEL, M. ; KNAPEK, S. ; ZUMBUSCH, G. ; TIMOTHY J. BARTH, D. E. R. M. D. R. T. S. Michael Griebel G. Michael Griebel (Hrsg.): *Numerical Simulation in Molecular Dynamics*. Springer, 2007

[58] GROPP, W. ; LUSK, E. ; THAKUR, R. : *Using MPI-2 Advanced Features of the Message-Passing Interface*. Cambridge, MA : MIT Press, 1999

[59] HAILE, J. M.: *Molecular Dynamics Simulation Elementary Methods*. JOHN WILEY SONS, INC., 1992

[60] HAN, K. ; FENG, Y. T. ; OWEN, D. R. J.: Coupled lattice Boltzmann and discrete element modelling of fluid-particle interaction problems. In: *Computer and Structures* 85 (2007), S. 1080–1088

[61] HENDRICKSON, B. ; DEVINE, K. : Dynamic Load Balancing in Computational Mechanics. In: *Comput. Methods Appl. Mech. Engrg.* 184 (2000), S. 485–500

[62] HENDRICKSON, B. ; LELAND, R. : A Multilevel Algorithm for Partitioning Graphs. In: *Proceedings of the Supercomputing '95, ACM*, December 1995

[63] HENDRICKSON, B. ; PLIMPTON, S. : Parallel Many-Body Simulations witwith All-to-All Communication. In: *Journal of Parallel and Distributed Computing* 27 (1995), S. 15–25

[64] HISHIURA, D. ; MATSUO, M. Y. ; SAKAGUCHI, H. : ppohDEM: Computational Performance for Open Source Code of the Discrete Element Method. In: *Computer Physics Communications* 185 (2014), S. 1486–1495

[65] HOPKINS, M. ; SONG, A. : Parallelization of DEM Models. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[66] HORTON, G. : A Multi-Level Diffusion Method for Dynamic Load Balancing. In: *Parallel Computing* 19 (1993), S. 209–218

[67] HOU, C. ; XU, J. ; WANG, P. ; HUANG, W. ; WANG, X. : Efficient GPU-accelerated Molecular Dynamics Simulation of Solid Covalent Crystals. In: *Computer Physics Communications* 184 (2013), S. 1364–1371

[68] HU, Y. F. ; BLAKE, R. J.: An Improved Diffusion Algorithm for Dynamic Load Balancing. In: *Parallel Computing* 25 (1999), S. 417–444

[69] HU, Y. F. ; BLAKE, R. J. ; EMERSON, D. R.: An Optimal Migration Algorithm for Dynamic Load Balancing. In: *Concurrency: Practice and Experience* 10(6) (1998), S. 467–483

[70] HWU, W.-M. W. ; CHIEF TODD GREEN, W.-M. W. H. E. (Hrsg.): *GPU Computing Gems Emeral Edition*. Elsevier, 2011

[71] INTEL: *ARK*. `http://ark.intel.com/`. – Accessed 10 July 2014

[72] INTEL: *Microprocessor Quick Reference Guide*. `http://www.intel.com/pressroom/kits/quickreffam.htm`. – Accessed 10 July 2014

[73] JABBARZADEH, A. ; ATKINSON, J. D. ; TANNER, R. I.: Parallel Simulation of Shear Flow of Polymers between Structured Walls by Molecular Dynamics Simulation on PVM. In: *Computer Physics Communications* 107 (1997), S. 123–136

[74] JABBARZADEH, A. ; ATKINSON, J. D. ; TANNER, R. I.: A Parallel Algorithm for Molecular Dynamics Simulation of Branched Molecules. In: *Computer Physics Communications* 150 (2003), S. 65–84

[75] JIA, W. ; FU, J. ; CAO, Z. ; WANG, L. ; CHI, X. ; GAO, W. ; WANG, L. W.: Fast Plane Wave Density Functional Theory Molecular Dynamics Calculations

on Multi-GPU Machines. In: *Journal of Computational Physics* 251 (2013), S. 102–115

[76] JING, L. ; STEPHANSSON, O. : *Fundamentals of Discrete Element Methods for Rock Engineering*. Elsevier, 2007

[77] JONES, M. T. ; PLASSMANN, P. E.: Computational Results for Parallel Unstructured Mesh Computations. In: *Computing Systems in Engineering* 5 (1994), S. 297–309

[78] KACIANAUSKAS, R. ; MAKNICKAS, A. ; KACENIAUSKAS, A. ; MARKAUSKAS, D. ; BALEVICIUS, R. : Parallel Discrete Element Simulation of Poly-dispersed Granular Material. In: *Advances in Engineering Software* 41 (2010), S. 52–63

[79] KAFUI, D. K. ; JOHNSON, S. ; THORNTON, C. ; SEVILLE, J. P. K.: Parallelization of a Lagrangian-Eulerian DEM/CFD Code for Application to Fluidized Beds. In: *Powder Technology* 207 (2011), S. 270–278

[80] KARAGIORGOS, G. ; MISSIRLIS, N. M.: Accelerated Diffusion Algorithms for Dynamic Load Balancing. In: *Information Processing Letters* 84 (2002), S. 61–67

[81] KARAKASIDIS, T. E. ; CHOLEVAS, N. S. ; LIAKOPOULOS, A. B.: Parallel short range molecular dynamics simulations on computer clusters: Performance evaluation and modeling. In: *Mathematical and Computer Modelling* 42 (2005), S. 783–798

[82] KARNIADAKIS, G. E. ; KIRBI II, R. : *Parallel Scientific Computing in C++ and MPI A seamless approach to parallel algorithms and their implementation*. Cambridge University Press, 2003

[83] KARYPIS, G. : *METIS 5.1.0: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Ordering of Sparse Matrices*. University of Minnesota, Department of Computer Science and Engineering, 2013

[84] KARYPIS, G. ; KUMAR, V. : A Coarse-Grain Parallel Multilevel k-way Partitioning Algorithm. In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997

[85] KARYPIS, G. ; KUMAR, V. : *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. Tech. Report CORR 95-035, University of Minnesota, Department of Computer Science, Minneapolis, MN, June 1995

[86] KARYPIS, G. ; SCHLOEGEL, K. : *ParMETIS 4.0: Parallel Graph Partitioning and Sparse Matrix Ordering Library*. University of Minnesota, Department of Computer Science and Engineering, 2013

[87] KENDALL, W. : *MPI Tutorial*. `http://mpitutorial.com/`. Version: June 2011. – Accessed June 2014

[88] KERNIGHAN, B. ; LIN, S. : An Efficient Heuristic Procedure for Partitioning Graphs. In: *Bell System Technical Journal* 29 (1970), Nr. 291–307

[89] KIRK, D. ; HWU, W.-m. W.: *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010

[90] KOMEIJI, Y. ; HARAGUCHI, M. ; NAGASHIMA, U. : Parallel Molecular Dynamics Simulation of a Protein. In: *Parallel Computing* 27 (2001), S. 977–987

[91] KRYSL, P. ; BELYTSCHKO, T. : Object-Oriented Parallelisation of Explicit Structural Dynamics with PVM. In: *Computers and Structures* 33 (1998), S. 259–273

[92] KYLASA, S. B. ; AKTULGA, H. M. ; GRAMA, A. Y.: PuReMD-GPU: A Reactive Molecular Dynamics Simulation Package for GPUs. In: *Journal of Computational Physics* 272 (2014), S. 343–359

[93] LE GRAND, S. ; GÖTZ, A. W. ; WALKER, R. C.: SPFP: Speed without Compromise-A Mixed Precision Model for GPU Accelerated Molecular Dynamics Simulations. In: *Computer Physics Communications* 184 (2013), S. 374–380

[94] LEACH, A. R.: *Molecular Modelling PRINCIPLES AND APPLICATIONS*. Prentice Hall, 2001

[95] LEI, Z. ; ROUGIER, E. ; KNIGHT, E. E. ; MUNJIZA, A. : A framework for grand scale parallelization of the combined finite discrete element method in 2d. In: *Computational Particle Mechanics* In Press (2014)

[96] LI, J. ; ZHOU, Z. ; SADUS, R. J.: Parallel Algorithms for Molecular Dynamics with Induction Forces. In: *Computer Physics Communications* 178 (2008), S. 384–392

[97] LI, X. ; CHU, X. ; SHENG, D. C.: A saturated discrete particle model and characteristic-based SPH method in granular materials. In: *International Journal for Numerical Methods in Engineering* 72 (2007), Nr. 7, S. 858–882

[98] LIN, W. : Mechanical Properties of Mesaverde Sandstone and Shale at High Pressures / Lawrence Livermore National Laboratory. 1983. – Forschungsbericht

[99] LIU, G. R. ; LIU, M. : *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. World Scientific Publishing Company, 2003

[100] LIU, H. ; TAFTI, D. K. ; LI, T. : Hybrid Parallelism in MFIX CFD-DEM Using OpenMP. In: *Powder Technology* 259 (2014), S. 22–29

[101] LIU, M. B. ; LIU, G. R. ; LAM, K. Y. ; ZONG, Z. : Smoothed particle hydrodynamics for numerical simulation of underwater explosion. In: *Computational Mechanics* 30 (2003), S. 106–118

[102] LIU, M. ; LIU, G. ; ZONG, Z. ; LAM, K. : Computer simulation of high explosive explosion using smoothed particle hydrodynamics methodology. In: *Computers & Fluids* 32 (2003), S. 305 – 322

[103] LIU, P. ; HRENYA, C. M.: Challenges of DEM: I. Competing Bottlenecks in Parallelization of Gas-Solid Flows. In: *Powder Technology* (2014)

[104] LIU, Q. ; YANG, J. ; LI, L. Y.: Numerical analysis of cold-formed sigma steel beams. In: MUNJIZA, A. (Hrsg.): *Discrete Element Methods, Simulatios of Discontinua: Theory and Application*, 2010, S. 199–204

[105] LU, J. ; ZHANG, J. ; WANG, X. ; WANG, L. ; GE, W. : Parallelization of Pseudo-Particle Modeling and its Application in Simulating Gas-Solid Fluidization. In: *Particuology* 7 (2009), S. 317–323

[106] MA, Z. ; FENG, C. ; LIU, T. ; LI, S. : An Optimized Algorithm for Discrete Element System Analysis Using CUDA. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[107] MAHABADI, O. K.: *Investigating the influence of micro-scale heterogeneity and microstructure on the failure and mechanical behaviour of geomaterials*, Graduate Department of Civil Engineering University of Toronto, Diss., 2012

[108] MAHABADI, O. K. ; GRASSELLI, G. ; MUNJIZA, A. : Y-GUI: A graphical user interface and pre-processor for the combined finite-discrete element code, Y2D, incorporating material heterogeneity. In: *Computers and Geosciences* 36 (2010), S. 241–252

[109] MAHABADI, O. K. ; LISJAK, A. : Application of FEMDEM to analyze fractured rock masses. In: *Accepted at DFNE 2014 conference*, 2014

[110] MAHABADI, O. K. ; LISJAK, A. ; GRASSELLI, G. ; MUNJIZA, A. : Y-Geo: a new combined finite-discrete element numerical code for geomechanical applications. In: *International Journal of Geomechanics, SPECIAL ISSUE: Advances in Modeling Rock Engineering Problems* 12 (2012), S. 676–688

[111] MEHRA, V. ; CHATURVEDI, S. : High velocity impact of metal sphere on thin metallic plates: A comparative smooth particle hydrodynamics study. In: *Journal of Computational Physics* 212 (2006), S. 318 – 337

[112] MILLER, G. L. ; TENG, S.-H. ; VAVASIS, S. A.: A Unified Geometric Approach to Graph Separators. In: *Proceedings of the 32nd Symposium on Foundations of Computer Science, IEEE, pp. 538-547*, 1991

[113] MINYARD, T. ; KALLINDERIS, Y. : Parallel Load Balancing for Dynamic Execution Environments. In: *Comput. Methods Appl. Mech. Engrg.* 189 (2000), S. 1295–1309

[114] MIO, H. ; HIGUCHI, R. ; ISHIMARU, W. ; SHIMOSAKA, A. ; SHIRAKAWA, Y. ; HIDAKA, J. : Effect of Paddle Rotational Speed on Particle Mixing Behavior in Electrophotographic System by Using Parallel Discrete Element Method. In: *Advanced Powder Technology* 20 (2009), S. 406–415

[115] MITCHELL, P. J. ; FINCHAM, D. : Multicomputer Molecular Dynamics. In: *Future Generation Computer Systems* 9 (1993), S. 5–10

[116] MÜLLER-PLATHE, F. : Parallelising a Molecular Dynamics Algorithm on a Multi-Processor Workstation. In: *Computer Physics Communications* 61 (1990), S. 285–293

[117] MUNJIZA, A. : *The Combined Finite Discrete Element Method*. Wiley, 2004

[118] MUNJIZA, A. ; ANDREWS, K. R. F.: Penalty function method for combined finite discrete element systems comprising large number of separate bodies. In: *International Journal for Numerical Methods in Engineering* 49 (2000), Nr. 11, S. 1377–1396

[119] MUNJIZA, A. ; ANDREWS, K. R. F. ; WHITE, J. K.: Combined single and smeared crack model in combined finite-discrete element analysis. In: *International Journal for Numerical Methods in Engineering* 44 (1999), S. 41–57

[120] MUNJIZA, A. ; ANDREWS, K. : NBS contact detection algorithm for bodies of similar size. In: *Int. J. Numer. Meth. Eng* 46 (1998)

[121] MUNJIZA, A. ; CARNEY, T. ; KNIGHT, E. ; SWIFT, R. P. ; GREENING, D. ; STEEDMAN, D. : The roots of possible chaoic behaviour in modelling and simulation of discontinua. In: *IASS-IACM 2008*, Internet-First University Press

[122] MUNJIZA, A. ; KNIGHT, E. ; E., R. : *Computational Mechanics of Discontinua*. Wiley, 2011

[123] MUNJIZA, A. ; LATHAM, J. P. ; ANDREWS, K. R. F.: Detonation gas model for combined finite-discrete element simulation of fracture and fragmentation. In: *International Journal for Numerical Methods in Engineering* 49 (2000), Nr. 12, S. 1495–1520

[124] MUNJIZA, A. ; OWEN, D. R. J. ; BICANIC, N. : A combined finite-discrete element method in transient dynamics of fracturing solids. In: *Eng. Comput.* 12 (1995), S. 145–174

[125] MUNSHI, A. ; GASTER, B. R. ; MATTSON, T. G. ; FUNG, J. ; GINSBURG, D. : *OpenCL Programming Guide*. Addison Wesley, 2011

[126] MUNSHI, A. (Hrsg.): *The OpenCL Specification, version 1.1, revision 44*. Khronos OpenCL Working Group, 2011

[127] MURTY, R. ; OKUNBOR, D. : Efficient Parallel Algorithms for Molecular Dynamics Simulations. In: *Parallel Computing* 25 (1999), S. 217–230

[128] NAKANO, A. ; VASHISHTA, P. ; KALIA, R. K.: Parallel Multiple-Time-Step Molecular Dynamics with Three-Body Interaction. In: *Computer Physics Communications* 77 (1993), S. 303–312

[129] NASSERI, M. H. B. ; MOHANTY, B. ; B., Y. : Fracture toughness measurements and acoustic emission activity in brittle rocks. In: *Pure Appl Geophys* 163 (2006), S. 917–945

[130] NAVARRO, C. A. ; HITSCHFELD-KAHLER, N. ; MATEU, L. : A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. In: *Communications in Computational Physics* 15 (2014), Nr. 2, S. 285–329

[131] NELSON, M. ; HUMPHREY, W. ; GURSOY, A. ; DALKE, A. ; KALE, L. ; SKEEL, R. ; SCHULTEN, K. : NAMD: a Parallel, Object-Oriented Molecular Dynamics Program. In: *International Journal of High Performance Computing Applications* 10 (1996), S. 251–268

[132] NGUYEN: *GPU Gems 3*. Addison Wesley Professional, 2007

[133] NISHIURA, D. ; SAKAGUCHI, H. : Parallel-Vector Algorithms for Particle Simulations on Shared-Memory Multiprocessors. In: *Journal of Computational Physics* 230 (2011), S. 1923–1938

[134] NVIDIA (Hrsg.): *OpenCL Programming Guide for the CUDA Architecture Version 3.1*. 2010

[135] ODEN, J. T. ; PATRA, A. ; FENG, Y. : Domain Decomposition for Adaptive HP Finite Element Methods. In: *Proceedings of the Seventh International Conference on Domain Decomposition Methods, State College, Pennsylvania*, October 1993

[136] OH, K. J. ; KLEIN, M. L.: A General Purpose Parallel Molecular Dynamics Simulation Program. In: *Computer Physics Communications* 174 (2006), S. 560–568

[137] OH, K. J. ; KLEIN, M. L.: A Parallel Molecular Dynamics Simulation Scheme for a Molecular System with Bond Constraints in NPT Ensemble. In: *Computer Physics Communications* 174 (2006), S. 263–269

[138] OWEN, D. R. J. ; FENG, Y. T.: Parallelised Finite/Discrete Element Simulation of Multi-fracturing Solids and Discrete Systems. In: *Engineering Computations* 18 (2001), S. 557–576

[139] OWEN, D. R. J. ; FENG, Y. T. ; HAN, K. ; PERIC, D. : Dynamic domain decomposition and load balancing in parallel simulation of finite/discrete elements. In: *ECCOMAS, Barcelona*, 2000, S. 11–14

[140] OWENS, J. D. ; HOUSTON, M. ; LUEBKE, D. ; GREEN, S. ; STONE, J. E. ; PHILLIPS, J. C.: GPU Computing. In: *Proceedings of the IEEE* 96 (2008), S. 879–899

[141] PACHECO, P. : *Parallel Programing with MPI*. Morgan Kaufmann, 1997

[142] PACHECO, P. : *An Introduction to Parallel Programming*. Morgran Kaufmann, 2011

[143] PATRA, A. ; ODEN, J. T.: Problem Decomposition for Adaptive HP Finite Element Methods. In: *J. Computing Systems Engrg.* 6 (1995)

[144] PATTERSON, D. A. ; HENNESSY, J. L.: *Computer Organization and Design: The Hardware/Software Interface, 4th Edition*. Morgan Kaufmann, 2009

[145] PILKINGTON, J. R. ; BADEN, S. B.: *Partitioning with Spacefilling Curves*. CSE Tech. Report CS94-349, Department of Computer Science and Engineering, University of California, San Diego, CA, 1994

[146] PLIMPTON, S. : Fast Parallel Algorithms for Short-Range Molecular Dynamics. In: *Journal of Computational Physics* 117 (1995), S. 1–19

[147] PLIMPTON, S. ; ATTAWAY, S. ; HENDRICKSON, B. ; SWEGLE, J. ; VAUGHAN, C. ; GARDNER, D. : Parallel Transient Dynamics Simulations: Algorithms for Contact Detection and Smoothed Particle Hydrodynamics. In: *Journal of Parallel and Distributed Computing* 50 (1998), S. 104–122

[148] PLIMPTON, S. ; HEFFELFINGER, G. : Scalable Parallel Molecular Dynamics on MIMD Supercomputers. In: *Proceedings of Scalable High Performance Computing Conference*, 1992

[149] PLIMPTON, S. ; HENDRICKSON, B. : A New Parallel Method for Molecular Dynamics Simulation of Macromolecular Systems. In: *Journal of Computational Chemistry* 17 (1996), S. 326–337

[150] POTHEN, A. ; SIMON, H. ; LIOU, K. : Partitioning Sparse Matrices with Eigenvectors of Graphs. In: *SIAM J. Matrix Anal.* 11 (1990), S. 430–452

[151] QIN, C.-Z. ; ZHAN, L. : Parallelizing Flow-accumulation Calculations on Graphics Processing Units-From Iterative DEM Preprocessing Algorithm to Recursive Multiple-flow-direction Algorithm. In: *Computers & Geosciences* 43 (2012), S. 7–16

[152] RABCZUK, T. ; EIBL, J. : Simulation of high velocity concrete fragmentation using SPH/MLSPH. In: *International Journal for Numerical Methods in Engineering* 56 (2003), S. 1421–1444

[153] RADEKE, C. A. ; GLASSER, B. J. ; KHINAST, J. G.: Large-Scale Powder Mixer Simulations Using Massively Parallel GPU Architectures. In: *Chemical Engineering Science* 65 (2010), S. 6435–6442

[154] RAJAMANI, R. K.: GPU High Performance Computing of Mill Charge Motion on a Desktop PC. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[155] RAPAPORT, D. C.: Multi-Million Particle Molecular Dynamics: I. Design Considerations for Vector Processing. In: *Computer Physics Communications* 62 (1991), S. 198–216

[156] RAPAPORT, D. C.: Multi-Million Particle Molecular Dynamics: II. Design Considerations for Distributed Processing. In: *Computer Physics Communications* 62 (1991), S. 217–228

[157] RAPAPORT, D. C.: Multi-Million Particle Molecular Dynamics: III. Design Considerations for Data-Parallel Processing. In: *Computer Physics Communications* 76 (1993), S. 301–317

[158] RAPAPORT, D. C.: Multibillion-Atom Molecular Dynamics Simulation: Design Considerations for Vector-Parallel Processing. In: *Computer Physics Communications* 174 (2006), S. 521–529

[159] RAPAPORT, D. : *The Art of MOLECULAR DYNAMICS SIMULATION Second Editon*. CAMBRIGE UNIVERSITY PRESS, 2004

[160] READ, J. (Hrsg.) ; STACEY, P. (Hrsg.): *Guidelines for Open Pit Slope Design*. CSIRO publishing, 2010

[161] REN, X. ; XU, J. ; QI, H. ; CUI, L. ; GE, W. ; LI, J. : GPU-based Discrete Element Simulation on a Tote Blender for Performance Improvement. In: *Powder Technology* 239 (2013), S. 348–357

[162] ROTARU, T. ; NAGELI, H.-H. : Dynamic Load Balancing by Diffusion in Heterogeneous Systems. In: *J. Parallel Distrib. Comput.* 64 (2004), S. 481–497

[163] ROUGIER, E. : *Discrete Element Method for Simulation of Gas Micro-Flows*, Queen Mary University Of London, Diss., 2009

[164] SADUS, R. J.: *Molecular Simulation of Fluids Theory, Algorithms and Object-Orientation*. Elsevier, 1999

[165] SANDERS, P. : Analysis of Nearest Neighbor Load Balancing Algorithms for Random Loads. In: *Parallel Computing* 25 (1999), S. 1013–1033

[166] SAWLEY, M. L. ; CLEARY, P. W.: A Parallel Discrete Element Method for Industrial Granular Flow Simulations. In: *EPFL Supercomput. Rev.* 11 (1999), S. 23–29

[167] SCHIAVA D'ALBANO, G. G.: *Computatinal and Algorithmic Solutions for Large Scale Combined Finite-Discrete Elements Simulations*, Queen Mary, University of London, Diss., 2014

[168] SCHIAVA D'ALBANO, G. G. ; MUNJIZA, A. : FEM/DEM Simulations on Multicore PC. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[169] SCHLOEGEL, K. ; KARYPIS, G. ; KUMAR, V. : Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes. In: *J. Parallel Distrib. Comput.* 47 (1997), S. 109–124

[170] SENGUPTA, S. ; HARRIS, M. ; GARLAND, M. ; OWENS, J. D.: Efficient Parallel Scan Algorithms for many-core GPUs. In: KURZAK, J. (Hrsg.) ; BADER, D. A. (Hrsg.) ; DONGARRA, J. (Hrsg.): *Scientific Computing with Multicore and Accelerators*. Taylor & Francis, 2011 (Chapman & Hall/CRC Computational Science), Kapitel 19, S. 413–442

[171] SHIGETO, Y. ; SAKAI, M. : Parallel Computing of Discrete Element Method on Multi-Core Processors. In: *Particuology* 9 (2011), S. 398–405

[172] SIMON, H. D.: Partitioning of Unstructured Problems for Parallel Processing. In: *Computing Systems in Engineering* 2 (1991), S. 135–148

[173] SMITH, W. : Molecular Dynamics on Hypercube Parallel Computers. In: *Computer Physics Communications* 62 (1991), S. 229–248

[174] SRINIVASAN, S. G. ; ASHOK, I. ; JONSSON, H. ; KALONJI, G. ; ZAHORJAN, J. : Dynamic-domain Decomposition Parallel Molecular Dynamics. In: *Computer Physics Communications* 102 (1997), S. 44–58

[175] STEUBEN, J. ; MUSTOE, G. ; TURNER, C. : Massively Parallel DEM Simulations with a Realistic Friction Model. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[176] STEUBEN, J. ; MUSTOE, G. ; TURNER, C. : Massively Parallel DEM Simulations with a Thermal Conduction Model. In: *Proceedings of 6th International Conference on Discrete Elements Methods and Related Techniques (DEM6), Golden*, 2013

[177] STIJNMAN, M. A. ; BISSELING, R. H. ; BARKEMA, G. T.: Partitioning 3D Space for Parallel Many-Particle Simulations. In: *Computer Physics Communications* 149 (2003), S. 121–134

[178] STONE, J. E. ; HARDY, D. J. ; UFIMTSEV, I. S. ; SCHULTEN, K. : GPU-accelerated Molecular Modeling Coming of Age. In: *Journal of Molecular Graphics and Modelling* 29 (2010), S. 116–125

[179] SUCCI, S. : *The Lattice Boltzmann Equation for Fluid dynamic and Beyond*. Oxford Sience Publications, 2001

[180] SUTTER, H. : The free lunch is over: A fundamental turn toward concurrency in software. In: *Dr. Dobb's Journal* 30 (2005)

[181] SUTTER, H. ; LARUS, J. : Software and the concurrency revolution. In: *Queue - Multiprocessors* 3 (2005), S. 54–62

[182] TAYLOR, V. E. ; STEVENS, R. L. ; ARNOLD, K. E.: Parallel Molecular Dynamics: Implications for Massively Parallel Machines. In: *Journal of Parallel and Distributed Computing* 45 (1997), S. 166–175

[183] TSUJI, T. ; YABUMOTO, K. ; TANAKA, T. : Spontaneous Structures in Three-Dimensional Bubbling Gas-Fluidized Bed by Parallel DEM-CFD Coupling Simulation. In: *Powder Technology* 184 (2008), S. 132–140

[184] VALDEZ-BALDERAS, D. ; DOMÍNGUEZ, J. M. ; ROGERS, B. D. ; CRESPO, A. J. C.: Towards Accelerating Smoothed Particle Hydrodynamics Simulations for Free-surface Flows on Multi-GPU Clusters. In: *Journal of Parallel and Distributed Computing* 73 (2013), S. 1483–1493

[185] VIDAL, Y. ; BONET, J. ; HUERTA, A. : Stabilized updated Lagrangian corrected SPH for explicit dynamic problems. In: *International Journal for Numerical Methods in Engineering* 69 (2007), S. 2687–2710

[186] VIOLEAU, D. ; ISSA, R. : Numerical modelling of complex turbulent free surface flows with the SPH method: an overview. In: *International Journal for Numerical Methods in Fluids* 53 (2007), S. 277–304

[187] VON NEUMANN, J. : First Draft of a Report on the EDVAC / Moore School of Electrical Engineering, University of Pennsylvania. 1945. – Forschungsbericht

[188] WALSHAW, C. ; CROSS, M. ; EVERETT, M. : A Localized Algorithm for Optimising Unstructured Mesh Partitions. In: *International Journal of Supercomputer Applications* 9 (2) (1995), S. 280–295

[189] WALSHAW, C. ; CROSS, M. ; EVERETT, M. : *Parallel Dynamic Graph-Partitioning for Unstructured Meshes*. Mathematics Research Report 97/IM/20, Centre for Numerical Modeling and Process Analysis, University of Greenwich, London, SE18 6PF, UK, March 1997

[190] WALSHAW, C. ; CROSS, M. ; EVERETT, M. G.: Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. In: *Journal of Parallel and Distributed Computing* 47 (1997), S. 102–108

[191] WANG, F. ; FENG, Y. T. ; OWEN, D. R. J. ; ZHANG, J. ; LIU, Y. : Parallel Analysis of Combined Finite/Discrete Element Systems on PC Cluster. In: *Acta Mechanica Sinica* 20 (2004), S. 534–540

[192] WANG, F. J. ; FENG, Y. T. ; OWEN, D. R. J.: Parallelisation for Finite-Discrete Element Analysis in a Distributed-Memory Environment. In: *International Journal of Computational Engineering Science* 5 (2004), S. 1–23

[193] WANG, L. ; LI, S. ; ZHANG, G. ; MA, Z. ; ZHANG, L. : A GPU-Based Parallel Procedure for Nonlinear Analysis of Complex Structures Using a Coupled FEM/DEM Approach. In: *Mathematical Problems in Engineering* 2013 (2013), S. 1–15

[194] WANG, X. ; GUO, L. ; GE, W. ; TANG, D. ; MA, J. ; YANG, Z. ; LI, J. : Parallel Implementation of Macro-Scale Pseudo-Particle Simulation for Particle-Fluid Systems. In: *Computers & Chemical Engineering* 29 (2005), S. 1543–1553

[195] WATTS, J. ; RIEFFEL, M. ; TAYLOR, S. : A Load Balancing Technique for Multiphase Computations. In: *Proceedings of the High Performance Computing '97, Society for Computer Simulation*, 1997, S. 15–20

[196] WHEAT, S. ; DEVINE, K. ; MACCABE, A. : Experience with Automatic, Dynamic Load Balancing and Adaptive Finite Element Computation. In: *Proceedings of the 27th Hawaii International Conference on System Sciences, IEEE*, January 1994, S. 463–472

[197] WILKINSON, B. ; ALLEN, M. : *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 2004. – ISBN 0131405632

[198] WILLEBEEK-LEMAIR, M. ; REEVES, A. P.: A Localized Dynamic Load Balancing Strategy for Highly Parallel Systems. In: *IEEE Transactions on Parallel and Distributed Systems* (1990), S. 380–383

[199] WILLEBEEK-LEMAIR, M. ; REEVES, A. P.: Strategies for Dynamic Load Balancing on Highly Parallel Computers. In: *IEEE Transactions on Parallel and Distributed Systems* 4 (1993), S. 979–993

[200] WILLIAMS, J. ; HOLMES, D. ; TILKE, P. : Muli-core Stragegies For Particle Methods. In: *Discrete Element Methods Simulations of Discontinua: Theory and Applications*, 2010

[201] WILLIAMS, R. D.: Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. In: *Concurrency: Practice and Experience* 3 (1991), Nr. 457-481

[202] WU, J.-S. ; HSU, Y.-L. ; LEE, Y.-M. : Parallel Implementation of Molecular Dynamics Simulation for Short-Ranged Interaction. In: *Computer Physics Communications* 170 (2005), S. 175–185

[203] WU, J.-S. ; ZHANG, H. ; DALRYMPLE, R. A. ; HÉRAULT, A. : Numerical Modeling of Dam-break Flood Through Intricate City Layouts Including Underground Spaces Using GPU-based SPH Method. In: *Journal of Hydrodynamics* 25 (6) (2013), S. 818–828

[204] XIONG, Q. ; LI, B. ; CHEN, F. ; MA, J. ; GE, W. ; LI, J. : Direct Numerical Simulation of Sub-Grid Structures in Gas-Solid Flow - GPU Implementation of Macro-Scale Pseudo-Particle Modeling. In: *Chemical Engineering Science* 65 (2010), S. 5356–5365

[205] XU, C. ; LAU, F. ; DIEKMANN, R. : *Decentralized Remapping of Data Parallel Applications in Distributed Memory Multiprocessors*. Tech. Report tr-rsfb-96-021, Department of Computer Science, University of Paderborn, Paderborn, Germany, September 1996

[206] XU, C. Z. ; LAU, F. C. M.: Analysis of the Generalized Dimension Exchange Method for Dynamic Load Balancing. In: *Journal of Parallel and Distributed Computing* 16 (1992), S. 385–393

[207] XU, D. ; KALIVIOTIS, E. ; MUNJIZA, A. ; AVITAL, E. ; JI, C. ; WILLIAMS, J. : Large scale simulation of red blood cell aggregation in shear flows. In: *Elsevier Journal of Biomechanics* 46 (2013), Nr. 11, S. 1810–1817

[208] XU, J. ; QI, H. ; FANG, X. ; LU, L. ; GE, W. ; WANG, X. ; XU, M. ; CHEN, F. ; HE, X. ; LI, J. : Quasi-real-time Simulation of Rotating Drum Using Discrete Element Method with Parallel GPU Computing. In: *Particuology* 9 (2011), S. 446–450

[209] ZHANG, L. ; QUIGLEY, S. F. ; CHAN, A. H. C.: A Fast Scalable Implementation of the Two-dimensional Triangular Discrete Element Method on a GPU Platform. In: *Advances in Engineering Software* 60-61 (2013), S. 70–80

[210] ZHENG, J. ; AN, X. ; HUANG, M. : GPU-based Parallel Algorithm for Particle Contact Detection and its Application in Self-compacting Concrete Flow Simulations. In: *Computers and Structures* 112-113 (2012), S. 193–204

[211] ZHENG, M. ; LI, X. ; GUO, L. : Algorithms of GPU-enabled Reactive Force Field (ReaxFF) Molecular Dynamics. In: *Journal of Molecular Graphics and Modelling* 41 (2013), S. 1–1

[212] ZIENKIEWICZ, O. ; TAYLOR, R. L.: *The Finite Element Method*. Butterworth Heinemann, 2000